Departamento de RNASA
Facultade de Informática
UNIVERSIDADE DA CORUÑA

TRABAJO FIN DE GRADO
GRADO EN INGENIERÍA INFORMÁTICA
MENCIÓN EN COMPUTACIÓN

# Desarrollo de redes de neuronas profundas para procesado de imagen.

**Estudiante:** Jose Manuel Silva (Caamaño)
**Director/a/es/as:** Julian Dorado (De la calle)
Marcos Gestal (Pose)

A Coruña, 6 de septiembre de 2019.

*Papá, mamá, gracias por ayudarme a cumplir este sueño.*

**Agradecementos**

En primer lugar, agradecer a los profesores Julián y Marcos por la ayuda y trato recibidos durante todo este trabajo. En segundo lugar, agradecer a familia y amigos todo el apoyo recibido durante estos años, tanto dentro como fuera del ámbito estudiantil.

**Resumen**

El TFG consiste en el desarrollo de redes de neuronas artificiales (RNA) para que realicen transformaciones en imágenes, de forma que se consiga una mejora de diferentes parámetros de la imagen dependiendo de las condiciones de esta.

Dentro de las RNA, hay unas arquitecturas específicas dentro del deep learning llamadas deep neural networks (DNN) que se caracterizan por tener un número alto de capas ocultas y de neuronas entre la capa de entrada y la de salida. En cada capa se procesan los datos mediante una serie de transformaciones lineales y no lineales hasta conseguir que los pesos de las neuronas tengan unos valores óptimos para que la salida producida y la esperada difieran muy poco.

Estas redes requieren de una carga computacional elevada y es por eso que se esta creando hardware especifico que aumente el rendimiento y que elimine la necesidad de grandes servidores que procesen los datos. Un ejemplo de este hardware es el Intel Movidius 2.

En este proyecto se plantea explorar el funcionamiento de un sistema hardware para implementación de RNA para poder desarrollar DNN en tiempos razonables de entrenamiento y ejecución. A partir de esto, construir conjuntos de entrenamiento con ejemplos de transformación de imagen, como por ejemplo HDRi, para poder entrenar DNN que permita mejorar significativamente futuras imágenes.

**Palabras clave:**

- TensorFLow
- Keras
- Redes Neuronales Profundas
- Procesado de imagen
- Intel Neural Compute Stick 2

# Índice general

# Índice de figuras

# Índice de cuadros

Capítulo 1

# Introducción

Durante los últimos cinco años, el auge de la Inteligencia Artificial (IA) ha resultado asombroso. Según un informe de la empresa Tractica [1] [2], es probable que crezca desde los actuales 643,7 millones de dólares estadounidenses hasta alcanzar los 36.000 millones en 2025. Esto quiere decir, que este mercado se multiplicaría por 57.

Y es que la IA tiene la capacidad de cambiar la manera en la que vivimos, pues nos puede ayudar a realizar tareas complejas en mucho menos tiempo e incluso solucionar problemas hasta ahora impensables. Un claro ejemplo son nuestros teléfonos móviles. Estos disponen de herramientas para conseguir mejores fotos, ordenarnos las imágenes por personas gracias a reconocimientos faciales e incluso poseen un altavoz inteligente para responder tus dudas.

El cambio más importante dentro de la IA reside en el hardware, el cual continúa mejorando según lo establecido en la ley de Moore [3]. Moore cayó en la cuenta de que el número de transistores en un circuito integrado se duplicaba cada año, lo que implicaba que la potencia de procesamiento se duplicara cada 18 meses.

Estas mejoras constantes han producido que nacieran empresas dedicadas a fabricar hardware especifico para IA como puede ser las unidades de procesamiento neuronal o NPU para dispositivos móviles. Estos chips son capaces de procesar mejor los procesos paralelos, secuenciar los pasos de los procesos y utilizar imágenes, voces y lenguaje natural en ellos. Utilizan muchos estímulos para resolver un problema y ofrecer una respuesta, en lugar de procesar la información de manera secuencial, que es lo que hace la CPU. Uno de estos fabricantes de hardware especifico es INTEL con su Intel Neural Compute Stick 2.

Este dispositivo esta diseñado para trabajar con un modelo computacional de IA que son las redes de neuronas artificiales, más concretamente con un tipo de redes llamadas redes neuronales profundas. Estas redes sobresalen en áreas como la detección de características o patrones en imágenes, ámbitos en donde la programación convencional resulta impensable en muchos casos. Un ejemplo que vemos a diario son los radares, que son capaces de reconocer los números y letras en las matriculas de los coches.

En nuestro caso la detección de estos patrones nos permitirá reconocer que zonas de las imágenes necesitan mas o menos intensidad para mejorar la iluminación y el contraste y por consiguiente mejorar la imagen.

Capítulo 2

# Fundamentos

## 2.1 Inteligencia Artificial

La Inteligencia Artificial (IA) [4] es la combinación de algoritmos planteados con el propósito de crear máquinas que presenten las mismas capacidades que el ser humano. La IA hace posible que las máquinas aprendan de la experiencia, se ajusten a nuevas aportaciones y realicen tareas como hacen los humanos. La mayoría de los ejemplos de inteligencia artificial de los que usted escucha hoy día – desde computadoras que juegan ajedrez hasta automóviles que se conducen por sí solos – se sustentan mayormente en aprendizaje profundo y procesamiento del lenguaje natural. Mediante el uso de estas tecnologías, las computadoras pueden ser entrenadas para realizar tareas específicas procesando grandes cantidades de datos y reconociendo patrones en los datos.

Los expertos en ciencias de la computación Stuart Russell y Peter Norvig diferencian varios tipos de inteligencia artificial [5]:

- **Sistemas que actúan como humanos:** se trata de computadoras que realizan tareas de forma similar a como lo hacen las personas. Es el caso de los robots.
- **Sistemas que piensan racionalmente:** intentan emular el pensamiento lógico racional de los humanos, es decir, se investiga cómo lograr que las máquinas puedan percibir, razonar y actuar en consecuencia. Los sistemas expertos se engloban en este grupo.
- **Sistemas que actúan racionalmente:** idealmente, son aquellos que tratan de imitar de manera racional el comportamiento humano, como los agentes inteligentes.
- **Sistemas que piensan como humanos:** automatizan actividades como la toma de decisiones, la resolución de problemas y el aprendizaje. Un ejemplo son las redes neuronales artificiales.

### 2.1.1 Redes Neuronales Artificiales

Una red neuronal artificial (RNA) [6] es un esquema de computación distribuida inspirada en la estructura del sistema nervioso de los seres humanos. La arquitectura de una red neuronal se forma conectando múltiples procesadores elementales (llamadas neuronas artificiales), siendo este un sistema adaptativo que posee un algoritmo para ajustar sus pesos (parámetros libres) para alcanzar los requerimientos de desempeño del problema basado en muestras representativas.

Por lo tanto podemos señalar que una RNA es un sistema de computación distribuida caracterizada por:

- Un conjunto de unidades elementales, cada una de las cuales posee bajas capacidades de procesamiento.
- Una densa estructura interconectada usando enlaces ponderados.
- Parámetros libres que son ajustados para satisfacer los requerimientos de desempeño.
- Un alto grado de paralelismo.

Las RNA se han hecho muy populares debido a la facilidad en su uso e implementación y la habilidad para aproximar cualquier función matemática. En la figura 2.1 podemos ver un esquema de como trabaja una RNA.

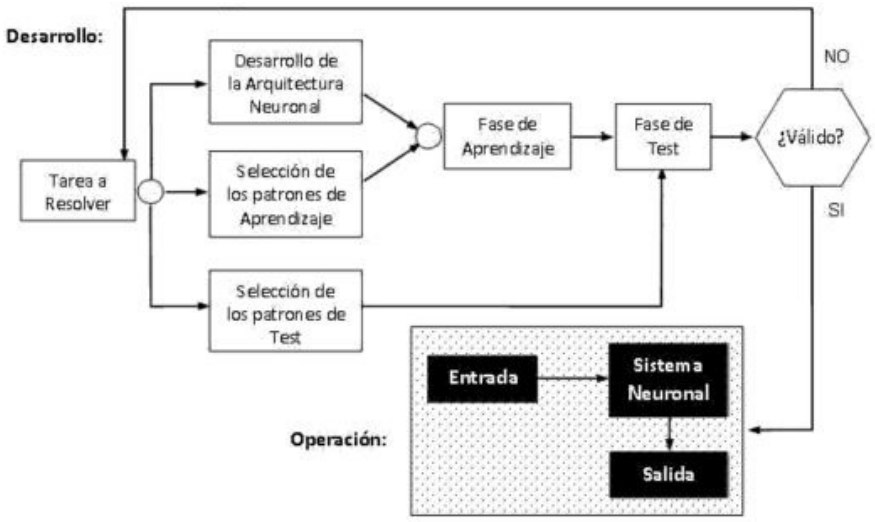



Figura 2.1: Esquema básico del trabajo con RNA

En una RNA existe un peso que es un valor numérico que pondera las señales que se reciben por sus entradas y que determina la fuerza de conexión entre 2 neuronas. Cuando se evalúa una neurona se debe calcular el conjunto de todas las fuerzas (denominado NET) que se reciben por sus entradas. Una vez calculado el valor conjunto de todas las entradas se aplica una función de activación(FA) que determinará el valor del estado interno de la neurona y que será lo que se transmita a su salida. La función de activación de una neurona es la encargada de relacionar la información de entrada de la neurona con el siguiente estado de activación que tenga esa neurona. Este esquema se puede ver en la figura 2.2.

La combinación de las señales que recibe una neurona se puede calcular como se muestra a continuación:

$$NET_i(t) = \sum_{j=1}^{N-1} [W_{ij} \cdot O_j \cdot (t-1)]$$

en donde Wij representa el peso de la conexión entre una neurona emisora j y neurona receptora i.

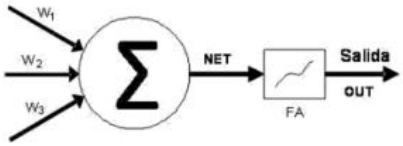



Figura 2.2: Neurona

Según la linealidad de las funciones que definen la red, las neuronas se pueden clasificar en:

- **Neuronas lineales**: Una neurona es lineal cuando su salida es linealmente dependiente de sus entradas, es decir, proporcional a las funciones de transferencia y de activación
- **Neuronas no lineales**: En estas neuronas, o bien la función de activación, o bien la función de transferencia (o ambas) son funciones no lineales, dando lugar a que la respuesta de la neurona no sea función lineal de sus entradas

En general, las neuronas suelen agruparse en unidades estructurales llamadas capas. Dentro de una capa, las neuronas suelen ser del mismo tipo. Se pueden distinguir tres tipos de capas:

- **De entrada:** reciben datos o señales procedentes del entorno.
- **De salida:** proporcionan la respuesta de la red a los estímulos de la entrada.
- **Ocultas:** no reciben ni suministran información al entorno (procesamiento interno de la red).

A continuación en la figura 2.3 podemos ver un esquema básico de la arquitectura de una red neuronal con una capa oculta.

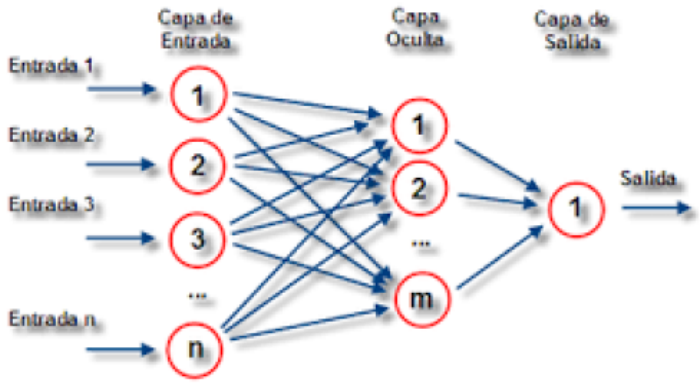



Figura 2.3: Esquema red neuronal

Según el flujo de los datos en las RNA se pueden dividir en dos grupos:

- Las redes alimentadas hacia delante, también conocidas como redes feedforward, son aquellas en las que la información se mueve en un único sentido, desde la entrada hacia la salida. Estas redes están organizadas en "capas". Cada capa agrupa a un conjunto de neuronas que reciben sinapsis de las neuronas de la capa anterior y emiten salidas hacia las neuronas de la capa siguiente. Entre las neuronas de una misma capa no hay sinapsis. El hecho de que no haya conexión entre las neuronas de una misma capa hace que no haya tiempos de espera. Se trata por tanto de redes rápidas en sus cálculos.
- Las redes con retroalimentación total o parcial pueden enviar estímulos a neuronas de capas anteriores, de su propia capa o a ellos mismos, por lo que desaparece el concepto de agrupamiento de las neuronas en capas. Cada neurona tendrá que calcular y recalcular su estado varias veces, hasta que todas las neuronas de la red alcancen un estado estable. Un estado estable es aquel en el que no ocurren cambios en la salida de ninguna neurona. No habiendo cambios en las salidas, las entradas de todas las neuronas serán constantes, por lo que no tendrán que modificar su estado de activación ni su respuesta, manteniéndose así un estado global estable

Es importante señalar que la propiedad más importantes de las redes neuronales artificiales es su capacidad de aprender a partir de un conjunto de patrones de entrenamientos, es decir, es capaz de encontrar un modelo que ajuste los datos. El proceso de aprendizaje, también conocido como entrenamiento de la red, puede ser supervisado, no supervisado o por refuerzo.

En el aprendizaje supervisado, los algoritmos trabajan con datos "etiquetados" (labeled data), intentado encontrar una función que, dadas las variables de entrada, les asigne la etiqueta de salida adecuada. El algoritmo se entrena con un conjunto etiquetado de datos y así aprende a asignar la etiqueta de salida adecuada a un nuevo valor, es decir, predice el valor da salida. Se suele usar en problemas de clasificación, como identificación de dígitos, diagnósticos, etc. También se usa en problemas de regresión, como predicciones meteorológicas, de expectativa de vida, de crecimiento, etc. La principal diferencia entre estos dos problemas (clasificación y regresión) es que la regresión tiene como objetivo predecir valores continuos (Números como el 1, 2.3, 3.1416 etc...), Y la clasificación tiene la tarea de asignar una clase, es decir predecir a que clase pertenece un conjunto de datos.

El aprendizaje no supervisado tiene lugar cuando no se dispone de datos "etiquetados" para el entrenamiento. Solo conocemos los datos de entrada, pero no existen datos de salida que correspondan a una determinada entrada. Por tanto, solo podemos describir la estructura de los datos, para intentar encontrar algún tipo de organización que simplifique el análisis.

El algoritmo de aprendizaje por refuerzo recibe algún tipo de valoración acerca de la idoneidad de la respuesta dada. Cuando la respuesta es correcta, el aprendizaje por refuerzo es similar al aprendizaje supervisado: en ambos casos reciben información acerca de lo que es apropiado. Sin embargo, ante las respuestas erróneas, en el supervisado se le dice qué debería haber respondido, mientras que el aprendizaje por refuerzo solo le informa acerca de que el comportamiento ha sido inapropiado y cuánto error se ha cometido.

#### 2.1.1.1 Redes Neuronales Profundas

Dentro de las RNA, hay unas arquitecturas específicas dentro del deep learning [7] llamadas redes neuronales profundas (DNN) (figura 2.4) que se caracterizan por tener un número alto de capas ocultas y de neuronas entre la capa de entrada y la de salida aunque no hay un umbral claro de profundidad que divida el aprendizaje superficial del aprendizaje profundo.

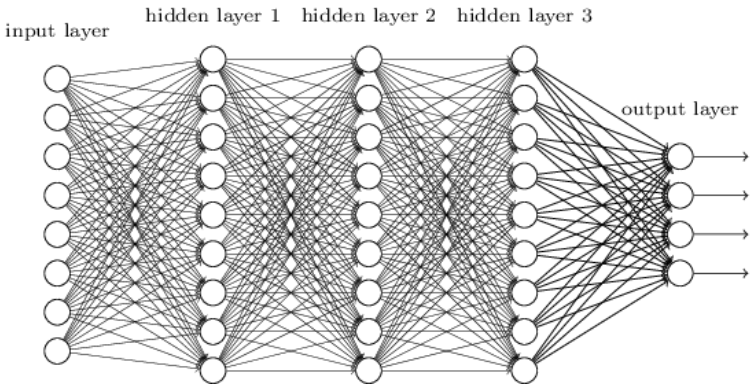



Figura 2.4: Red neuronal profunda

La primera capa de la red neuronal procesa una entrada de datos como por ejemplo una imagen, y la pasa a la siguiente capa como salida, este proceso se va repitiendo sucesivamente hasta completar todas las capas de la red neuronal.

Estas redes terminan en una capa de salida: un clasificador logístico que asigna una probabilidad a un resultado o etiqueta en particular. Esto quiere decir que, dados unos datos sin procesar en forma de una imagen, una DNN puede decidir, por ejemplo, que los datos de entrada tienen una probabilidad del 90 por ciento de representar a una persona.

El objetivo del entrenamiento es hacer que el costo del entrenamiento sea lo más pequeño posible a través de millones de ejemplos de entrenamiento. Para hacer esto, la red ajusta los pesos y sesgos hasta que la predicción coincida con el resultado correcto.

Existen varios tipos de DNN especializados en diferentes aspectos:

- Redes de Boltzman restringida(RBN) o autoencoders: es una red de dos capas poco profunda en la que la primera es la visible y la segunda la oculta. Cada nodo en la capa visible está conectado a cada nodo en la capa oculta. La red se conoce como restringida porque no se permite que dos capas dentro de la misma capa compartan una conexión. RBN es el equivalente matemático de un traductor bidireccional. Un pase hacia adelante toma entradas y las traduce en un conjunto de números que codifica las entradas. Mientras tanto, una pasada hacia atrás toma este conjunto de números y los traduce de nuevo en entradas reconstruidas.

- Redes de creencias profundas (DBN): se forman combinando RBM con un método de entrenamiento inteligente. DBN se puede visualizar como una pila de RBM donde la capa oculta de un RBM es la capa visible de la RBM que se encuentra arriba. El primer RBM está capacitado para reconstruir su entrada con la mayor precisión posible. La capa oculta de la primera RBM se toma como la capa visible de la segunda RBM y la segunda RBM se entrena utilizando las salidas de la primera RBM. Este proceso se itera hasta que cada capa en la red está entrenada. Son buenas para reconocimiento de imágenes.

- Redes Generativas antagónicas (GANs): Las GANs son redes neuronales profundas que comprenden dos redes, enfrentadas una con la otra, de ahí el nombre de "antagónicas". Una de las redes se conoce como generador y es la encargada de generar nuevas instancias de datos. La otra red se denomina discriminador y es la que evalúa las salidas del generador para determinar su autenticidad. Se puede enseñar a los GAN a crear mundos paralelos similares a los nuestros en cualquier dominio: imágenes, música, habla, prosa.

- Redes neuronales recurrentes (RNN): son redes neuronales en las que los datos pueden fluir en cualquier dirección. Estas redes se utilizan para aplicaciones como el modelado de lenguaje o el procesamiento de lenguaje natural (NLP). Se denominan recurrentes ya que repiten la misma tarea para cada elemento de una secuencia, y la salida se basa en los cálculos anteriores. Por lo tanto, se puede decir que las RNN tienen una "memoria" que captura información sobre lo que se ha calculado previamente. Las RNN pueden usar información en secuencias muy largas, pero solo pueden mirar hacia atrás unos pocos pasos.

- Redes neuronales profundas convolucionales (CNN): La idea detrás de las redes neuronales convolucionales es la idea de un "filtro en movimiento" que pasa a través de la imagen. Este filtro en movimiento, o convolución, se aplica a cierta vecindad de nodos que, por ejemplo, pueden ser píxeles, donde el filtro aplicado es 0.5 x el valor del nodo. Si tenemos una imagen, que no es mas que una matriz de píxeles, las características aprendidas en la primera capa pueden ser, por ejemplo, la aparición o no de ejes en una parte concreta de la imagen, en la segunda capa detectaría uniones de ejes y en la tercera capa aprendería combinaciones que correspondería a partes de objetos. Lo más característico de este método es que estas capas realizan el descubrimiento de características sin intervención humana, aprendiéndolo directamente de los datos brutos. Cuanto más avance en la red neuronal, más complejas serán las características que sus nodos pueden reconocer.

#### 2.1.1.2 Autoencoders

Los autoencoders [8] son redes neuronales que generan nuevos datos, primero comprimiendo la entrada en un espacio de variables latentes y luego reconstruyendo la salida basandose en la información comprimida(figura 2.5). El objetivo de un autoencoder es aprender una representación (codificación) para un conjunto de datos, entrenando la red a través de operaciones y filtros reduciendo su tamaño para ir extrayendo nuevas características, como ya explicamos en el apartado de RNN. A diferencia de estas existe una fase posterior a la reducción y es la reconstrucción. Es aquí donde el autoencoder intenta generar una representación lo más cercana posible a su entrada original, de ahí su nombre. Para ello pasamos los mismos filtros en sentido inverso para recuperar el tamaño inicial.

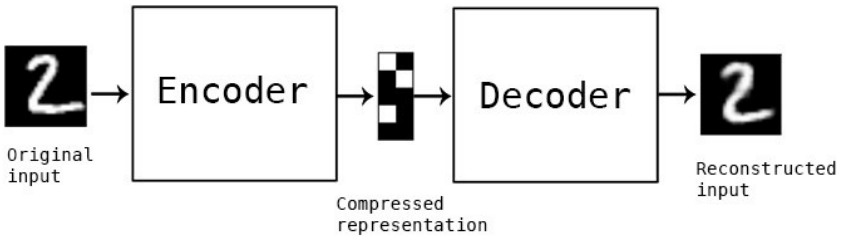



Figura 2.5: Autoencoder

Este tipo de red consta de dos partes:

- Encoder: la parte de la red que comprime la entrada en un espacio de variables latentes y que puede representarse mediante la función de codificación h = f (x).
- Decoder: la parte que trata de reconstruir la entrada basandose en la información recolectada previamente. Se representa mediante la función de decodificación r = g (h).

Existen varios tipos de autoencoders:

- Vanilla autoencoder
- Multilayer autoencoder
- Convolutional autoencoder
- Regularized autoencoder
- Denoising autoencoder (DAE)
- Autoencoder contractivo (CAE)
- Autoencoder variacional (VAE)

En nuestro caso nos centraremos en los autoencoders convolucionales o convolutional autoencoder.

#### 2.1.1.3 U-Net

Las U-net [9] son redes neuronales convolucionales cuya arquitectura se asemeja mucho a la arquitectura del autoencoder: Tenemos una primera parte que comprime los datos y los codifica, y una segunda parte que decodifica los datos para reconstruir la imagen. La diferencia reside en que las U-net disponen de conexiones entre las capas del codificador y decodificador que están enfrentadas.

Estas nuevas conexiones entre capas se encargan de ayudar en la reconstrucción de los datos comprimidos, enviándole información del contexto para construir la imagen resultante con mayor resolución en la salida. Para predecir los píxeles en la región del borde de la imagen, el contexto que falta se extrapola reflejando la imagen de entrada.

Como podemos ver en la figura 2.6, donde cada capa del lado de la izquierda esta unido a la de la derecha. Cada uno de los colores de las flechas representa un proceso ya sea convolución (conv), copiar y recortar (copy and crop), reducir tamaño (max pool) o aumentar tamaño (up-conv).

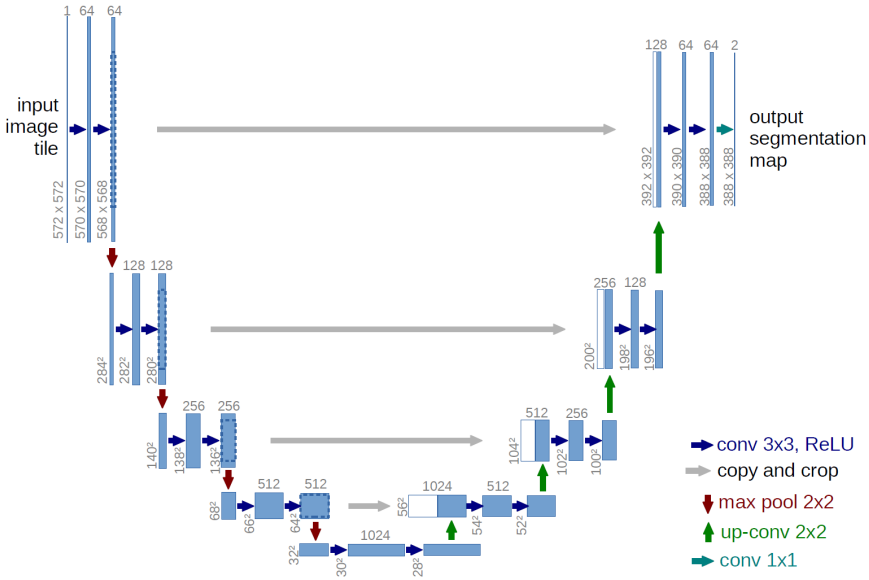



Figura 2.6: Ejemplo U-net para 32x32 píxeles

Cada cuadro azul corresponde a un mapa de características multicanal. El número de canales se indica en la parte superior del cuadro. El tamaño xy se proporciona en el borde inferior izquierdo de la caja. Los cuadros blancos representan mapas de características copiados. Las flechas denotan las diferentes operaciones.

### 2.1.2 Herramientas

#### 2.1.2.1 TensorFlow

TensorFlow [10] es una biblioteca de software de código abierto, desarrollada por Google para computación numérica, que utiliza gráficos de flujo de datos. Los nodos en las gráficas representan operaciones matemáticas, mientras que los bordes de las gráficas representan las matrices de datos multidimensionales (tensores) comunicadas entre ellos.

Los modelos matemáticos utilizados son redes neuronales, que en función de la arquitectura de capas y neuronas que la conforman se podrá modelizar desde un simple modelo de regresión hasta una arquitectura mucho más compleja de machine learning.

TensorFlow es una gran plataforma para construir y entrenar redes neuronales, que permiten detectar y descifrar patrones y correlaciones, análogos al aprendizaje y razonamiento usados por los humanos.

Además dispone integración con distintos sistemas operativos y compatibilidad con distintos procesadores gráficos. API para Python, C++, Java, C#, JavaScript, etc.

#### 2.1.2.2 Keras

Keras [11] es una librería de Python que proporciona, de una manera sencilla, la creación de una gran gama de modelos de Deep Learning usando como ”backend” otras librerías como TensorFlow, Theano (biblioteca de Python y compilador de optimización para manipular y evaluar expresiones matemáticas) o CNTK (marco de aprendizaje profundo que describe las redes neuronales como una serie de pasos computacionales a través de un gráfico dirigido).

Ofrece un conjunto de abstracciones más intuitivas y de alto nivel haciendo más sencillo el desarrollo de modelos de aprendizaje profundo independientemente del backend computacional utilizado. Además del soporte para las redes neuronales estándar, Keras ofrece soporte para las Redes Neuronales Convolucionales y para las Redes Neuronales Recurrentes, así como combinaciones de las dos.

Keras, permite una creación de prototipos fácil y rápida, a través de la facilidad de uso minimizando el número de acciones de usuario requeridas, de la modularidad, la extensibilidad añadiendo módulos fácilmente y fácil de depurar.

En 2017, el equipo de TensorFlow de Google decidió ofrecer soporte a Keras en la biblioteca core de TensorFlow, a partir de la versión 2.0.

## 2.2 Hardware para IA

Constantemente leemos noticias sobre el desarrollo de nuevos algoritmos y tecnologías de IA como procesamiento de voz, traducción instantánea, reconocimiento de imágenes, manipulación de vídeos, etc; pero las opciones para aplicar esa tecnología en nuestro día a día son aún bastante limitadas.

La IA está demandando nuevo hardware más potente y específicamente diseñado para las cargas de trabajo y cálculos diferentes que se requieren hoy en día, como son predicción, inferencia o intuición. El valor de este hardware en el aprendizaje profundo a menudo se pasa por alto. Las redes neuronales prácticas y eficientes, que se han investigado durante los últimos 50 años, eran inviables sin hardware para admitir algoritmos de aprendizaje profundo. Muchos logros de IA fueron posibles gracias a los avances en hardware, y es que el hardware es la base de todo lo que puede hacer el software.

### 2.2.1 Intel Neural Compute Stick 2

Una unidad de procesamiento de visión (VPU) es una clase emergente de microprocesador. Es un tipo específico de acelerador de IA diseñado para acelerar las tareas de visión artificial. El Intel NCS 2 [12] está alimentado por la última generación de Intel VPU: el Intel Movidius Myriad X VPU. Es el primero en presentar un motor de cálculo neuronal: un acelerador de inferencia de red neuronal de hardware dedicado que ofrece un rendimiento adicional. En la figura 2.7 (páxina 14) podemos ver un ejemplo de su arquitectura. Combinado con la distribución Intel del kit de herramientas OpenVINO que admite más redes, el Intel NCS 2 ofrece a los desarrolladores una mayor flexibilidad de creación de prototipos. Además, gracias al Intel AI: En el ecosistema ”In production”, los desarrolladores ahora pueden trasladar sus prototipos Intel NCS 2 a otros factores de forma y producir sus diseños.

#### 2.2.1.1 Intel Movidius Myriad X

La Unidad de procesamiento de visión Intel Movidius Myriad X [13] es la generación más reciente de Intel VPU. Incluye 16 potentes núcleos de procesamiento (llamados núcleos SHAVE) y un acelerador de hardware de redes neuronales profundas dedicado para aplicaciones de inferencia en inteligencia artificial y visión de alto desempeño, todo con bajo consumo de energía.

La arquitectura proporciona un enfoque modular para configurar las cargas de trabajo de la imagen y la visión ya que combina un conjunto de aceleradores de hardware de imagen y visión, como su motor de cálculo neuronal, con una serie de procesadores vectoriales VLIW programables en C, todos accediendo a un sistema común de chip-memoria (figura 2.8).

Es un salto gigante sobre la generación anterior de VPU. Puede alcanzar hasta 105 FPS (80 típicos) y puede realizar más de 1 billón de operaciones de punto flotante por segundo como un acelerador de red neuronal dedicado. Lo mejor de todo es que Myriad X es una VPU de potencia ultra baja, que necesita muy poca energía para realizarla.

Este enfoque permite el procesamiento de señal de imagen (ISP) sin la necesidad de realizar viajes a la memoria para obtener la mejor eficiencia de energía, todo con una metodología de flujo de datos que reduce la potencia al minimizar el movimiento de datos. Las VPU de Movidius ofrecen una variedad óptima entre la capacidad de programación y el rendimiento a baja potencia.

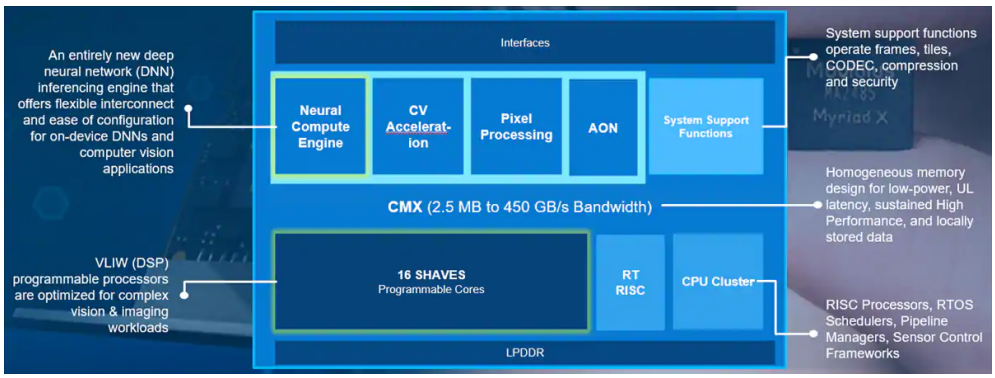



Figura 2.7: Arquitectura Intel Compute Stick 2

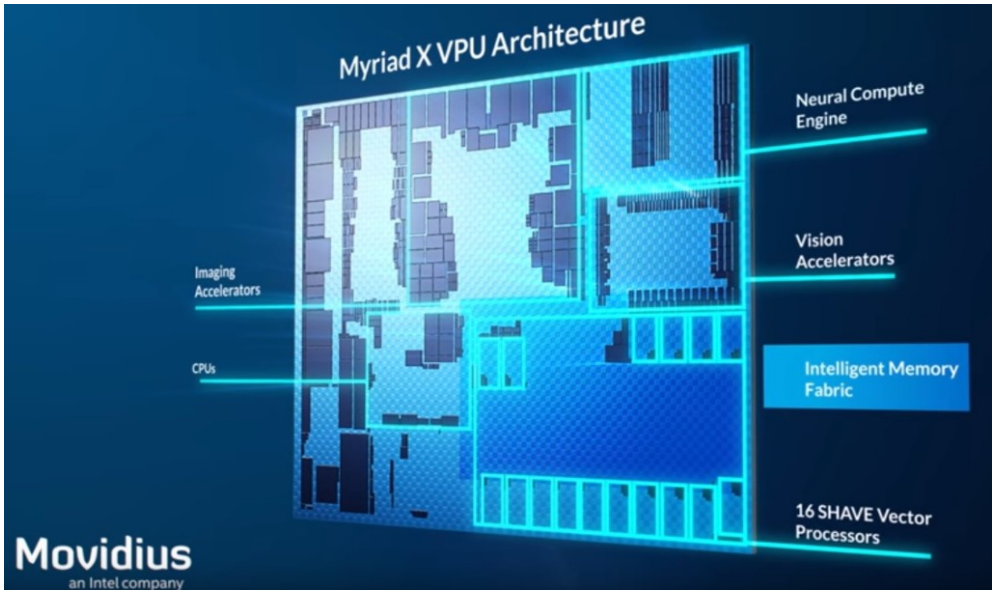



Figura 2.8: Arquitectura Myriad X

#### 2.2.1.2 Especificaciones técnicas

- **Procesador:** Unidad de procesamiento de visión Intel Movidius Myriad X con 16 núcleos de SHAVE programables
- **Marcos compatibles:** TensorFlow, Caffe, MXNet, Kald y ONNX
- **Conectividad:** USB 3.0 Tipo A
- **Dimensiones:** 2,85 pulg. x 1,06 pulg. x 0,55 pulg. (72,5 mm x 27 mm x 14 mm)
- **Resolución de video:** Codificación basada en hw con una resolución de hasta 4K
- **Temperatura de funcionamiento:** 0° C a 40° C
- **Sistemas operativos compatibles:** Ubuntu 16.04 LTS, CentOS 7.4 y Windows 10

#### 2.2.1.3 Diferencias con la versión anterior

Como se puede ver en la figura 2.9, el Intel NCS 2 ofrece hasta ocho veces más desempeño, en comparación con la Barra de cómputo neuronal Intel Movidius (NCS) de la generación anterior.

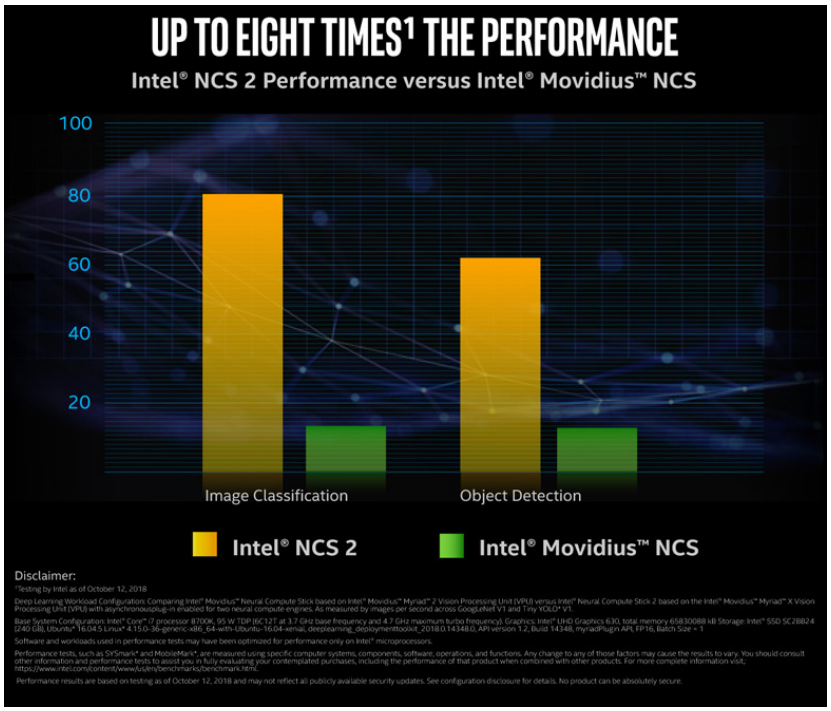



Figura 2.9: Diferencia con la versión anterior

## 2.3 Mejora de Imagen

Cuando hablamos de mejorar una imagen nos referimos al proceso por el cual apreciamos mejor lo que se representa en ella, ya sea mejorando la nitidez o haciendo que se vean mejor partes que son inapreciables, y este es el punto en el que vamos a centrarlos. Para ello las mejoras se centraran en aumentar la iluminación y el contraste de las imágenes en las zonas mas oscuras, intentando que así se aprecien mejor las formas que hay en ellas.

### 2.3.1 HDRi

HDRi (imágenes de alto rango dinámico) son un conjunto de técnicas que permiten un mejor rango dinámico (El rango dinámico es la diferencia entre la luz más clara y la oscuridad más oscura de una imagen) de luminancias entre las zonas más claras y las más oscuras de una imagen.

El ojo humano, mediante la adaptación del iris y otros métodos, se ajusta constantemente para adaptarse a un rango más amplio de luminancia presente en el entorno. Las imágenes HDR representan con más exactitud que el ojo humano el extenso rango de niveles de intensidad encontrados en escenas reales.

Figura 2.10: Ejemplo HDR con 3 imágenes

La técnica más común que utilizan las cámaras hoy en día para capturar imágenes HDR se llama bracketing u horquillado. Esta técnica consiste en tomar varias imágenes del mismo

sitio variando entre cada una de ellas uno o varios parámetros de la exposición, como puede ser el enfoque, la apertura del objetivo, u otros. Después, estas imágenes se combinan en una sola produciendo una imagen con un alto rango dinámico. En la figura 2.10 podemos ver un ejemplo en el que se tomaron 3 imágenes con distintas amplitudes (3 de abajo) y se combinaron para generar la imagen HDR (de arriba).

Para una mayor calidad en las imágenes, se recomienda que al realizar el horquillado todas las imágenes sacadas sean desde la misma posición y apuntando al mismo lugar, si no se producirán errores a la hora de intentar juntarlas.

### 2.3.2 Ajuste de curvas

Para entender como funciona el ajuste de curvas primero tenemos que saber lo que es el histograma de una imagen. El Histograma es un gráfico que representa la cantidad de píxeles de cada uno de los 256 niveles de iluminación, desde el valor 0 (Negro absoluto) al 255 (Blanco absoluto), de una imagen (figura 2.11).

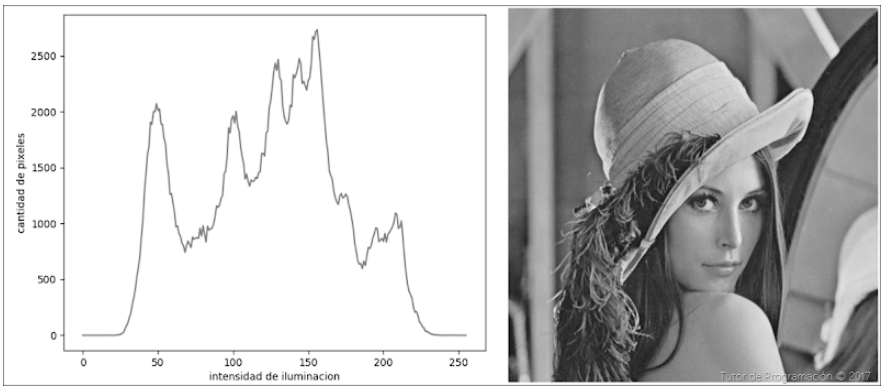



Figura 2.11: Histograma de una imagen en blanco y negro

En el caso de imágenes a color, tendríamos un histograma por cada canal, es decir, un histograma para el rojo, otro para el azul y otro para el verde.

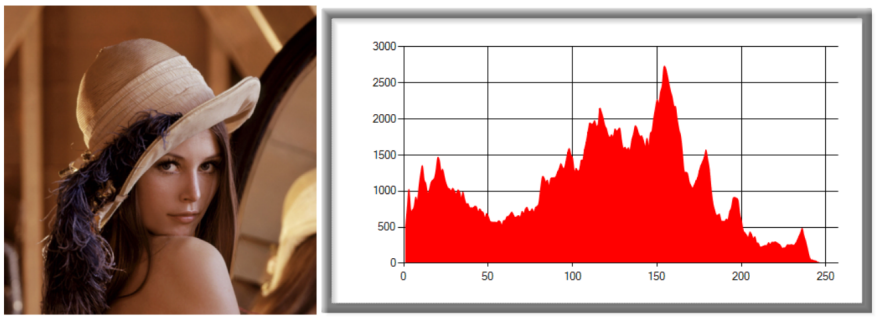



Figura 2.12: Histograma del rojo de una imagen

Los cambios de brillo y contraste que le produzcamos a una imagen, son cambios que se verán reflejados en el histograma. Un ejemplo lo podemos ver en la figura 2.13, que tras aplicarle una curva de aumento de luminosidad al histograma de la izquierda, nos da como resultado el histograma de la derecha, en el que podemos observar como se ha estirado hacia los tonos mas claros (derecha).

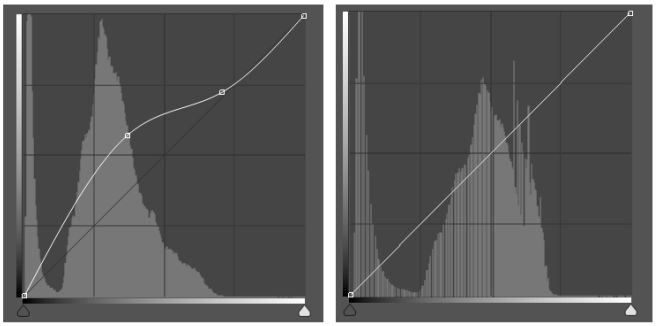


Figura 2.13: Cambio en histograma al aplicar curva aumento luminosidad

Otra manera de cambiar el histograma es asignar mediante una función, valores nuevos de intensidad a los valores antiguos de una imagen. Esto se representa como una línea recta diagonal si no hay ningún cambio (figura 2.14). El eje horizontal representa el rango de información de la imagen de negros a blancos (histograma), es la información de entrada que manipularemos. El eje vertical representa la escala tonal a la que podemos llevar esa información que ingresamos, es la información de salida que nos dará el resultado final. Las secciones más elevadas de la curva representan áreas de mayor contraste, mientras que las más bajas representan áreas de menor contraste.

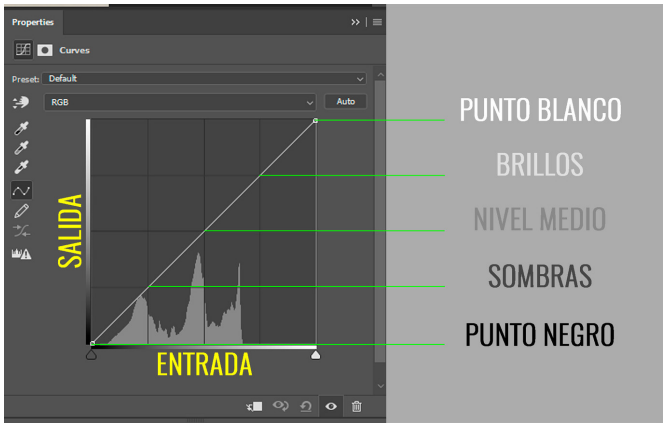


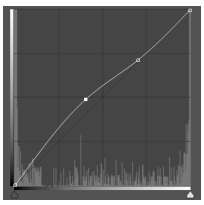

Figura 2.14: Puntos clave de las curvas

Si trabajamos con un color en concreto (rojo, verde o azul) y aumentamos la diagonal agregaremos más de ese color primario y si la bajamos estaremos agregando más de su color

complementario. Dependiendo de si subes la curva en la zona de las sombras, los medios tonos o las altas luces, ese color se aplicará tan solo en esa zona (figura 2.15).


Figura 2.15: Zonas de las curvas

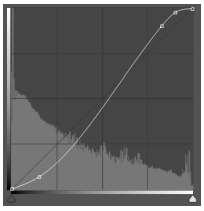
Si movemos un punto de la parte superior de la curva, ajustamos las iluminaciones; si movemos un punto de la parte central de la curva, ajustamos los medios tonos; y si movemos un punto de la parte inferior de la curva, ajusta las sombras. Para oscurecer las iluminaciones, movemos hacia abajo un punto cercano a la parte superior de la curva. Si movemos un punto hacia abajo o hacia la derecha se asignará el valor de entrada a un valor de salida más bajo, y la imagen se oscurecerá. Para aclarar las sombras, movemos hacia arriba un punto cercano a la parte inferior de la curva. Si movemos un punto hacia arriba o hacia la izquierda asignará un valor de entrada más bajo a un valor de salida más alto, y la imagen se aclarará.



(a) Curva iluminación

(b) Imágenes con la curva aplicada

Figura 2.16: Curvas de iluminación aplicada a imagenes



(a) Curva contraste

(b) Imágenes con la curva aplicada

Figura 2.17: Curvas usadas en el entrenamiento

Capítulo 3

# Estado de la cuestión

En este apartado se explicaran varias herramientas, como son U-GAT-IT, Photolemur y SRCNN, cuyo procedimiento nos puede ayudar a conseguir nuestro objetivo dado que fueron creadas para realizar tareas similares a lo que buscamos.

Primero se realizará una explicación detallada con ejemplos de cada una de las herramientas y al final, en un apartado de consideraciones, una breve comparación con las herramientas este proyecto.

## 3.1 U-GAT-IT

U-GAT-IT [14] es una red que produce traducciones superiores entre imágenes. La traducción de imagen a imagen, en la que las características estilísticas de una imagen se imponen en el contenido de otra para crear una nueva imagen, se había limitado a traducir formas o texturas. Esta nueva red traduce ambos creando una salida visualmente más satisfactoria.

Un ejemplo de esta aplicación podemos verlo en la siguiente figura 3.1 donde transforma imágenes de caras en bocetos de animes, imágenes de caballos en cebras o gatos en perros.

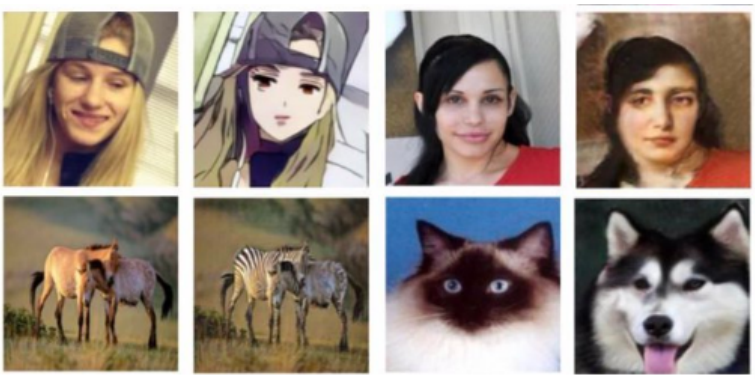


Figura 3.1: Ejemplo U-GAT-IT

A diferencia de otras redes donde las redes de traducción de imagen a imagen, U-GAT-IT funciona mejor con estilos de imagen particulares dado que agrega capas que lo hacen útil en una variedad de estilos. Para ello agrega una capa de atención que guía a nuestro modelo para enfocarse en regiones más importantes y que pondera la importancia de cada mapa de características en función del estilo de cada imagen. Además, disponen de la nueva función AdaLIN (Adaptive Layer-Instance Normalization) que ayuda a nuestro modelo guiado por la atención a controlar de manera flexible la cantidad de cambio en la forma y la textura mediante parámetros aprendidos en función de los conjuntos de datos.

U-GAT-IT utiliza una arquitectura típica de GAN: un discriminador clasifica las imágenes como reales o generadas y un generador intenta engañar al discriminador.

El generador (figura 3.2) toma las imágenes y utiliza una CNN para extraer mapas de características que codifican formas y texturas, junto con una capa de ponderación intermedia que aprende la importancia de cada mapa de características. Estos mapas ponderados se pasan a la capa de atención para evaluar las correspondencias de píxeles, y el generador produce una imagen a partir de ahí.

El discriminador(figura 3.3) toma la primera imagen como un ejemplo de estilo del mundo real y la segunda como candidato en el mismo estilo que es real o generado. Al igual que el generador, codifica ambas imágenes para presentar mapas a través de una CNN y utiliza una capa de ponderación para guiar la capa de atención. El discriminador clasifica la imagen candidata en función de la salida de la capa de atención.

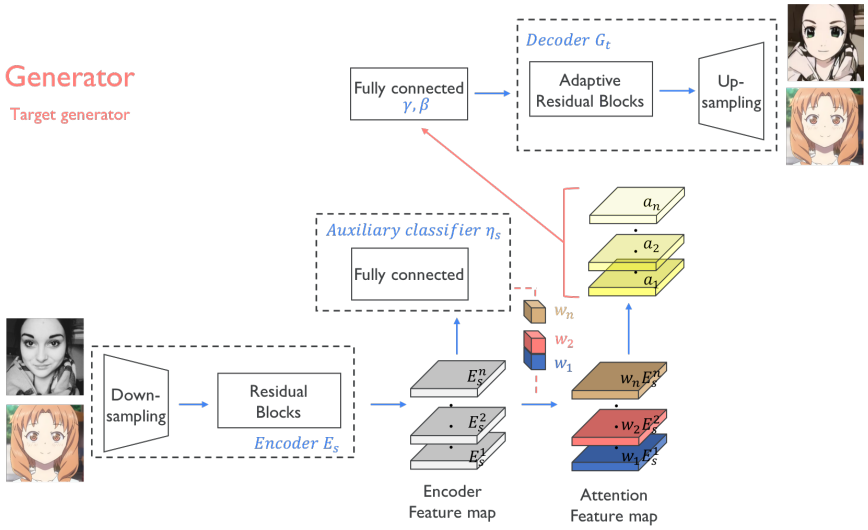



Figura 3.2: Generador U-GAT-IT

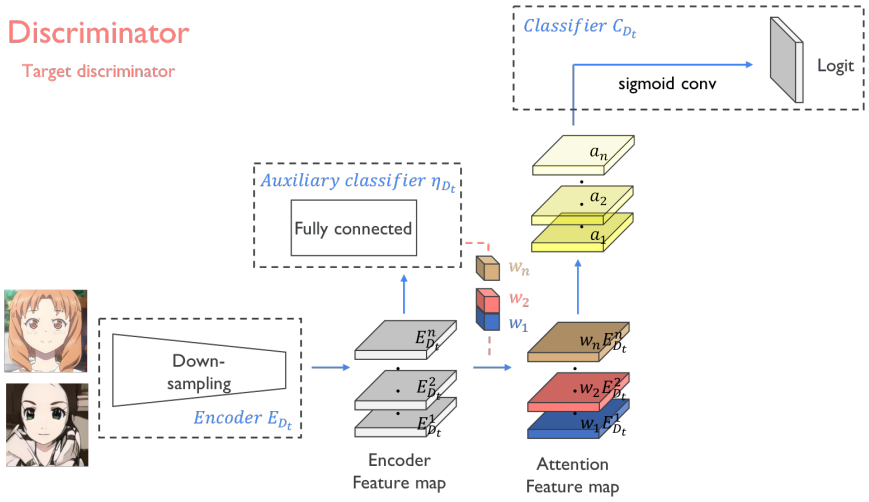



Figura 3.3: Discriminador U-GAT-IT

Las redes de deepfake de mejor rendimiento se basan en gran medida en las técnicas de traducción de imágenes.

## 3.2 Photolemur

Otro ejemplo de sistema que modifica imágenes mediante RNA es Photolemur [15]. Este sistema permite utilizar fotografías tanto en formato RAW como JPG y una vez cargada la imagen que queramos el programa analiza en la fotografía diversos aspectos tales como la exposición, el enfoque, los elementos reconocibles como el mar, el cielo, montañas, caras, etcétera. También distingue entre retratos, paisajes y macro fotografías y aplica diferentes algoritmos en función de lo que haya reconocido.

Una vez la imagen ha sido procesada, Photolemur nos ofrece la pantalla dividida en dos partes con una raya vertical que podemos mover para ir cubriendo y descubriendo la zona procesada y la original y ver el efecto que ha conseguido Photolemur en nuestra imagen.

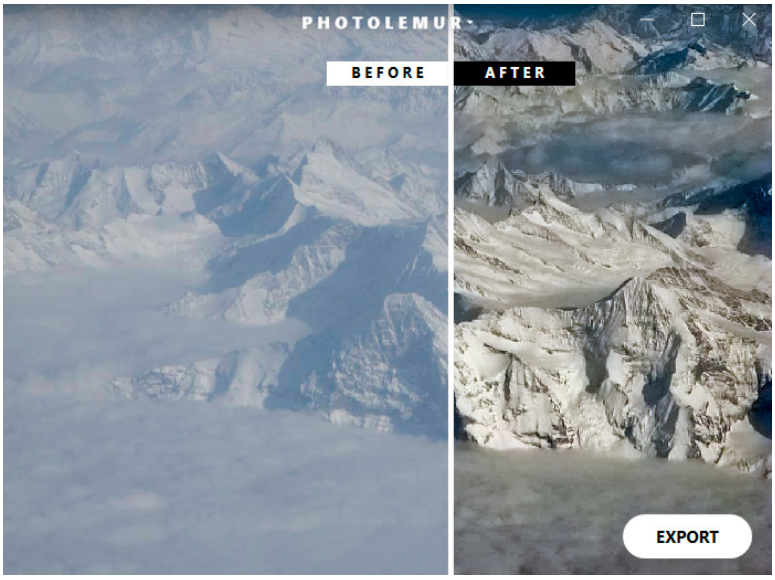



Figura 3.4: Ejemplo Photolemur

## 3.3 SRCNN

La Red neuronal convolucional de súper resolución (SRCNN) [16] se utiliza para la súper resolución de imagen única (SR), que es un problema clásico en la visión por computadora. Con un mejor enfoque de SR, podemos obtener de una imagen pequeña, una imagen de tamaño mas grande con una buena calidad sin difuminar.

La red SRCNN se caracteriza por ser una red no profunda. Su funcionamiento se divide en 3 partes: extracción y representación de parches, mapeo no lineal y reconstrucción, tal y como se muestra en la figura 3.5.

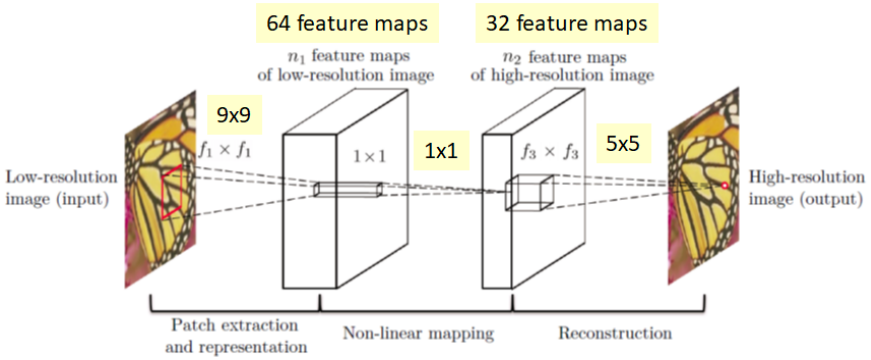



Figura 3.5: Red SRCNN

En la fase de extracción y representación de parches escalaremos la entrada de baja resolución al tamaño deseado mediante la interpolación bicúbica(haciendo un promedio de Z

pixeles adyacentes). Para ello introduciremos una capa de convolución de c x Z × Z × n1, donde c es el número de canales de la imagen(3), X es el tamaño del filtro(9) y n1 es el número de filtros(64).

En el siguiente paso haremos un mapeo no lineal, el cual se realiza al igual que en el paso anterior con una capa convolucional, pero en este caso de n1 x 1 x 1 x n2, donde n2 es el nuevo numero de filtros. Al reducir el numero de filtros conseguimos mapear los vectores de baja resolución a vectores de alta resolución.

Para reconstruir la imagen usaremos una ultima capa convolucional de n2 x Y x Y x c, donde Y será el tamaño del filtro, en este caso de 5.

Para medir la calidad de la reconstrucción en el ámbito de las imágenes se usa la Proporción Máxima de Señal a Ruido o PSNR. Esta medida expresa generalmente en decibelios y cuanto mayor sea su valor, mejor será la reconstrucción.

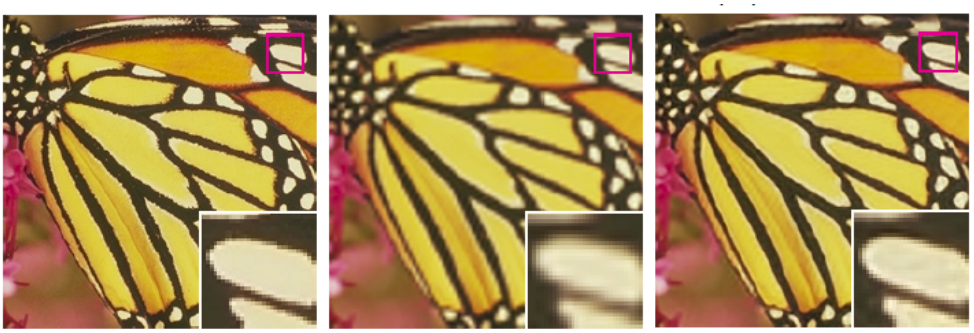



Figura 3.6: Red SRCNN

## 3.4 Consideraciones

Desde el punto de vista de nuestro sistema, U-GAT-IT es un sistema mucho más complejo en el que se irían generando imágenes hasta conseguir que la salida aprendiera la curva que queremos. Es aquí donde casi todos estos algoritmos tienen el principal inconveniente, el tiempo. Con el sistema de U-GAT-IT, cada vez que queramos enseñarle algo, o mejorar los ejemplos, deberemos volver a entrenar toda la red. En el caso de Photolemur, al usar varios algoritmos distintos para mejorar la imagen, requiere de mucho tiempo para poder procesarla.

Otro punto en contra de U-GAT-IT es que usa bloques residuales (Residual blocks) en el generador. En este sistema cada capa alimenta a la siguiente capa y directamente a las capas a unos 2-3 saltos de distancia hacia delante. En nuestro caso nos interesa más tener conexiones directas entre las capas enfrentadas, puesto que lo que queremos el mantener la forma de la imagen inicial y que la reconstrucción sea lo mas parecida a la imagen inicial, cosa que en este algoritmo no sucede, dado que buscan una transformación de la imagen.

En el caso de SRCNN, es un algoritmo que nos sería útil para intentar mejorar la calidad y definición de las imágenes, pero no esta diseñado para hacer mejoras en la imagen más allá de una mejora en la resolución. Este algoritmo no requiere de un entrenamiento para que aprenda, realiza un trabajo procedimental y sería útil para mejorar ciertos algoritmos en los que se pierda definición en la salida.

En este proyecto, lo que buscamos es un sistema rápido y flexible en el que la mejora producida en la imagen se realice en cuestión de segundos. Para ello, necesitamos un sistema que no sea muy complejo y que no tenga muchas capas de profundidad, pues esto ralentizaría el proceso de mejora. Además, buscamos un sistema que sea capaz de mejorar cualquier tipo de imagen, ya sea paisajes, objetos o rostros, sin restringirlo a un tipo solo de imagen.

# Capítulo 4

# Sistema

## 4.1 Sistema

Se va a desarrollar un sistema que realice una mejora en imágenes, basado en Deep learning, y que sea rápido y portable mediante el uso de un hardware especializado y de bajo coste.

Para nuestro sistema necesitaremos generar una red entrenada que reciba imágenes y nos devuelva la misma imagen mejorada, transformar ese modelo a un modelo valido para nuestro hardware y hacer que funcione en este.

Para generar la red entrenada usaremos el lenguaje de programación de Python, que junto con las librerías de TensorFlow y Keras, nos proporcionará un entorno de trabajo sencillo sobre el que crear nuestra red y manipularla a nuestro gusto.

La arquitectura de la red será un autoencoder que entrenaremos con imágenes de entrada-salida a las que le aplicamos una mejora en la salida para que la red aprenda esta mejora para futuras imágenes.

La mejora sistemática de imágenes se realizará mediante un aumento de la iluminación y el contraste por zonas, aumentando la visibilidad en las zonas más oscuras y permitiendo distinguir cosas que hasta ahora eran inapreciables.

Una vez dispongamos de nuestro modelo entrenado, el siguiente paso es adaptarlo al hardware especializado, en nuestro caso es el Intel Compute Stick 2 (NCS2).

Para ello, dado que usamos Tensorflow, hay que realizar unos ajustes previos para que el NCS2 sea capaz de leer los pesos y conexiones y no de problemas. Este paso se llama congelar el modelo y viene explicado en el apartado 4.6.1.1.

Por último, integrar nuestro sistema en el NCS2.

El procedimiento usado se dividirá en los siguientes pasos:

- Proceso de creación y entrenamiento de la red
- Mejora de las imágenes
- Pruebas
- Resultados
- Integración en el NCS2

## 4.2 Creación y entrenamiento de la red

Antes de la creación del modelo, se realizó un estudio de compatibilidad de las diferentes librerías que se podrían usar y que serían admitidas por el hardware especifico que hemos elegido (especificadas en el apartado 4.6). Además, este hardware no admite todo tipo de capas, por lo que también hubo que hacer un estudio sobre esto.

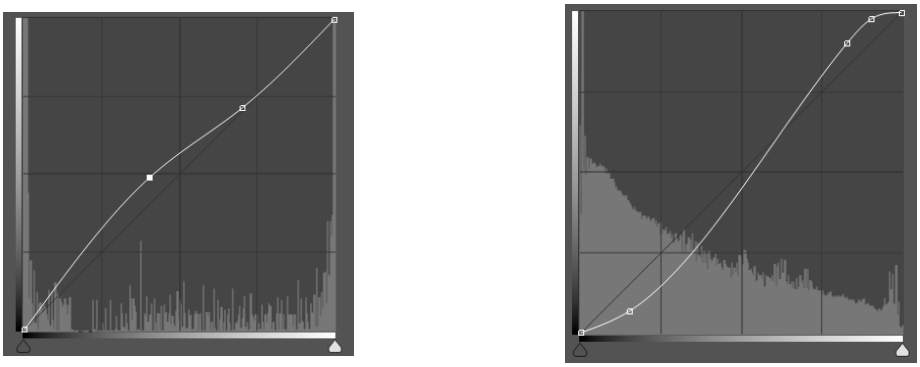
Una vez se decidió trabajar con Tensorflow y Keras y dado que queremos que la salida de nuestra red tenga el mismo tamaño que la entrada, se realizó otro estudio sobre diferentes arquitecturas en las que respetase el tamaño de entrada en la salida. Así mismo, necesitamos que nuestro sistema produzca una salida en función de la entrada, es decir, respetando la representación de los objetos que hay en ella, sin alterar su posición, producir una salida mejorada, por lo que necesitamos una arquitectura que construya la salida usando información de la imagen de entrada.

## 4.3 Mejora de imagen

El proyecto consiste en la mejora sistemática de imágenes mediante un aumento de la iluminación y el contraste por zonas, aumentando la visibilidad en las zonas más oscuras y permitiendo distinguir cosas que eran inapreciables. Un ejemplo similar a estas mejoras es HDRi.

El primer paso es reunir el conjunto de imágenes entrada-salida que usaremos para entrenar el modelo (explicado en la sección 4.3.1).

Para el conjunto de salida, cogeremos las mismas imágenes aplicándoles una curva de intensidad 4.1a y una curva de contraste 4.1b que aumentarían la iluminación y el contraste en las zonas más oscuras de la imagen. Para ello se establecieron 3 conjuntos de salida: salida con mejora en iluminación, salida con mejora en contraste y una salida en la que se le aplicó contraste e iluminación.



(a) Curva iluminación

(b) Curva contraste

Figura 4.1: Curvas usadas en el entrenamiento

En el caso de la curva de intensidad, se escogió esta curva porque aumenta la luminosidad en las zonas mas oscuras y medios tonos y mantiene la luminosidad en las zonas más claras, que es lo que buscamos. En cuanto a la curva de contraste, se escogió esta porque aumenta el contraste de la imagen haciendo a los colores mas cercanos al negro más oscuros y a los colores cercanos al blanco mas blancos, produciendo cambios de tonalidad mas grandes al tener una pendiente más empinada, que es lo que buscamos.

Los puntos de la figura 4.1a se encuentran en los puntos (103,125) y (180,185). En el caso de la figura 4.1b tendremos los puntos (38,17), (212,231) y (231,250).

### 4.3.1 Imágenes de entrada

Las imágenes de entrada serán de tamaño 1024x1024 a color y en escala de grises.

Las imágenes deberán abarcar toda la gama de colores e intensidades posibles para que nuestra red aprenda a tratar cualquier color e intensidad que se encuentre. Para ello se han creado 10 imágenes con figuras de distintas formas y tamaños, haciendo un degradado de oscuro a claro en diferentes direcciones, ya sea a color o en escala de grises, de forma que se abarque el mayor numero de colores e intensidades posibles. Estas imágenes, junto con otras 20 reales, serán nuestro conjunto de entrada.

Para la fase de entrenamiento se han utilizado 20 de las 32 imágenes (las 10 creadas mas 10 reales) y se han reservado 2 de las imágenes reales para validación. Las 8 imágenes restantes se han usado para testear.

Cabe destacar que las imágenes llamadas ”de salida” que están a continuación son las imágenes de salida usadas en el entrenamiento, dado que, como mencionamos anteriormente, nuestra red se entrena con imágenes de entrada-salida a las que se les aplicó una mejora (curva). Estas imágenes son distintas de las imágenes de salida de la red una vez ya esta entrenada, que son los resultados del apartado 4.5 usadas para testear la red.

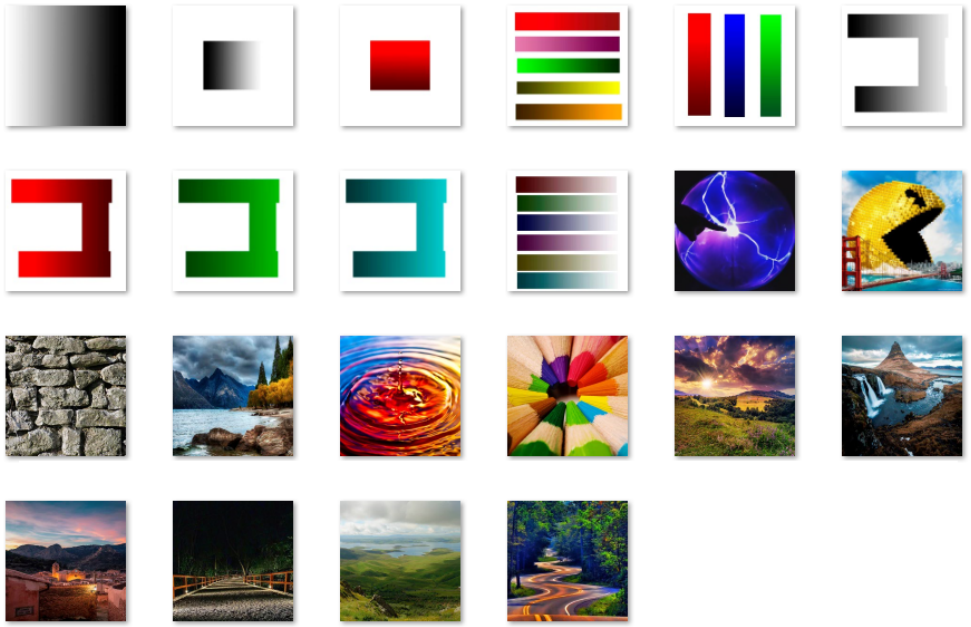


Figura 4.2: Imágenes de entrada

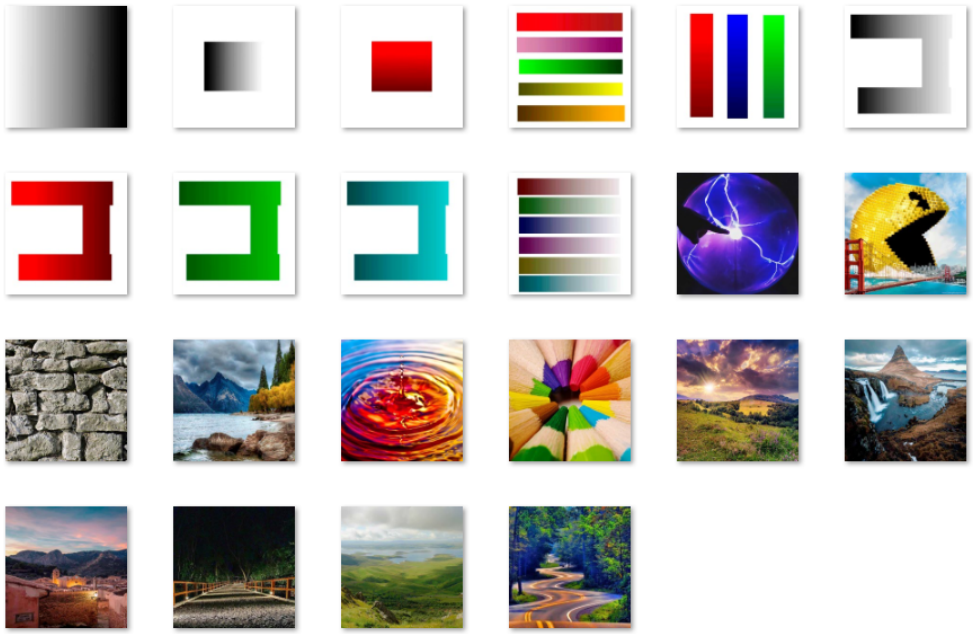


Figura 4.3: Imágenes de salida con curva de iluminación aplicada

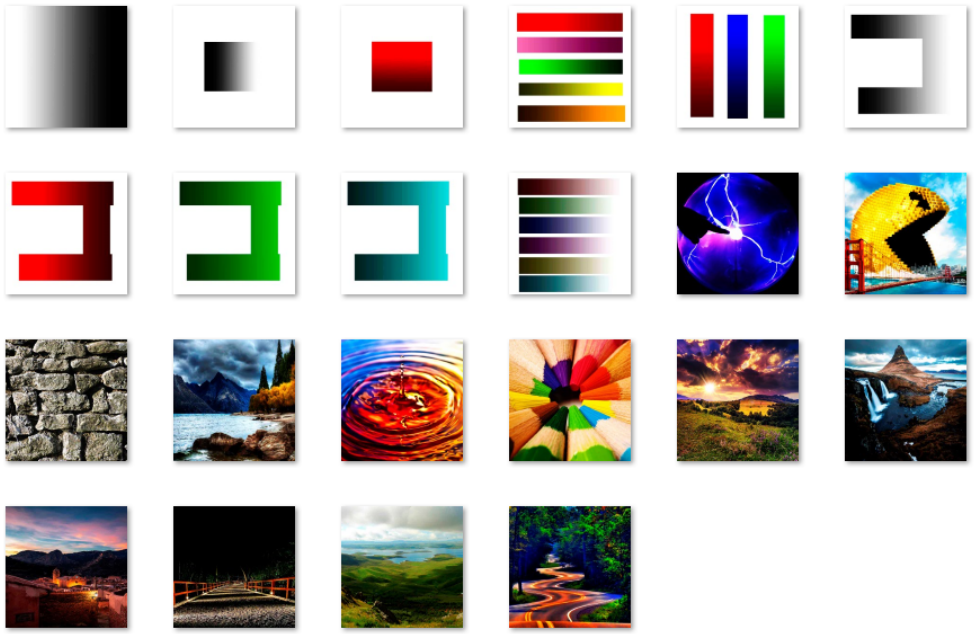


Figura 4.4: Imágenes de salida con curva de contraste aplicada

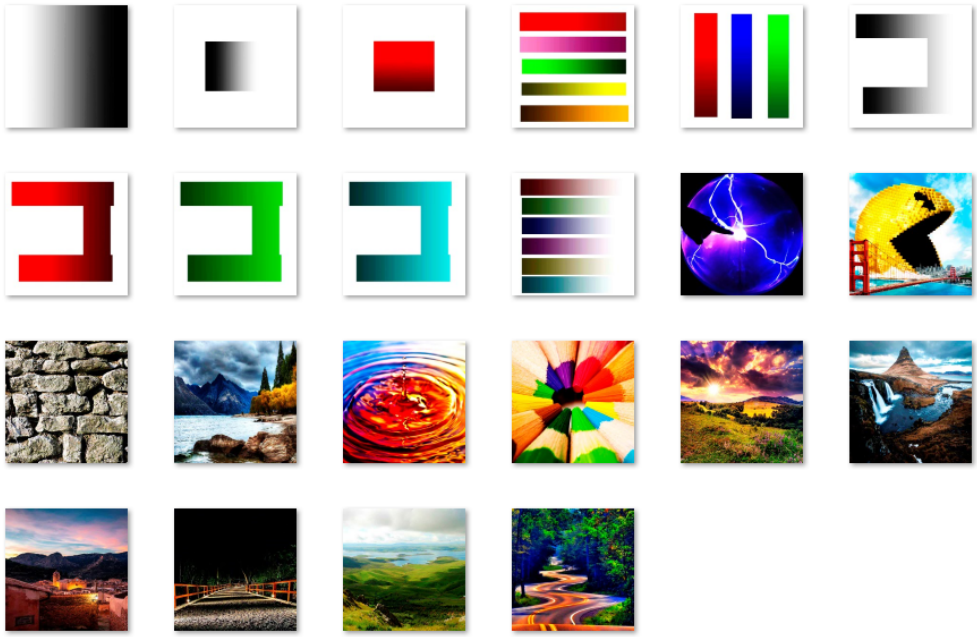


Figura 4.5: Imágenes de salida con curva de iluminación y contraste aplicada

## 4.4 Pruebas

Las pruebas se realizaron sobre un autoencoder (se ha decidido utilizar esta arquitectura por las causas explicadas en el punto 4.2) al que se le irán pasando imágenes de distinto tamaño y se le irán añadiendo nuevos componentes (como puede ser un clasificador) y capas para ver de que forma aprende mejor las curvas que queremos enseñarle y conseguir los mejores resultados.

Al ser un procedimiento incremental, se realizarán pruebas cada vez que el tamaño de la imagen cambie o se añada un componente nuevo.

Siempre que se realice algún ajuste de las imágenes de entrada, estas tendrán que respetar el mismo tamaño y orden en las imágenes de salida, de forma que si las entradas pasan a ser trozos de 16x16 píxeles, la salida tendrá que ser del mismo tamaño y representando el mismo trozo que la imagen de entrada. En estos casos en los que se dividan las imágenes en trozos, usaremos un 10% de esas imágenes para validación

Las imágenes usadas para las pruebas (entradas 4.6 y salidas deseadas 4.7) son las siguientes:

Figura 4.6: Imágenes de test

Figura 4.7: Imágenes de salida de test ideales (con ambas curvas aplicadas)

Las salidas de cada una de las siguientes pruebas estarán en el apartado de resultados 4.5.

1. La primera prueba es con un autoencoder sencillo de 11 capas. La primera capa es la capa de entrada, en la que especificaríamos el tamaño de las imágenes de entrada. En nuestro caso las imágenes son de 1024x1024x3 (el x3 viene dado por los 3 canales de colores RGB).

   Las siguientes 5 capas componen en codificador. Como se puede apreciar en la figura 4.8 tenemos capas de convolución y capas de reducción (o MaxPooling) en donde iremos codificando y reduciendo el tamaño de la imagen.

   Las ultimas 5 capas corresponden al decodificador, y no son mas que una copia del codificador invertida. En este caso usaremos la función UpSampling para aumentar el tamaño de las imágenes.

   La ultima de las 5 capas será la que proporcione la salida, en este caso otra imagen de tamaño 1024x1024x3.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_7 (InputLayer)         (None, 1024, 1024, 3)     0
_________________________________________________________________
conv2d_37 (Conv2D)           (None, 1024, 1024, 16)    160
_________________________________________________________________
max_pooling2d_13 (MaxPooling (None, 512, 512, 16)      0
_________________________________________________________________
conv2d_38 (Conv2D)           (None, 512, 512, 8)       1160
_________________________________________________________________
max_pooling2d_14 (MaxPooling (None, 256, 256, 8)       0
_________________________________________________________________
conv2d_39 (Conv2D)           (None, 256, 256, 8)       584
_________________________________________________________________
conv2d_40 (Conv2D)           (None, 256, 256, 8)       584
_________________________________________________________________
up_sampling2d_13 (UpSampling (None, 512, 512, 8)       0
_________________________________________________________________
conv2d_41 (Conv2D)           (None, 512, 512, 16)      1168
_________________________________________________________________
up_sampling2d_14 (UpSampling (None, 1024, 1024, 16)    0
_________________________________________________________________
conv2d_42 (Conv2D)           (None, 1024, 1024, 3)     145
=================================================================
Total params: 3,801
Trainable params: 3,801
Non-trainable params: 0
_________________________________________________________________
```

Figura 4.8: Autoencoder de 11 capas con imagenes de 1024x1024

2. La siguiente prueba fue comprobar si el entrenamiento se realiza mejor separando los canales de la imagen. Así, dividiríamos cada imagen en sus 3 respectivos canales y las iríamos apilando en una lista, una detrás de otra, de la forma: r, g, b, r, g, b, r, etc. Lo mismo para las imágenes de salida. De esta forma, como se puede ver en la figura 4.9 la entrada y la salida de la red pasarian a ser arrays de 1024x1024x1.

| Layer (type) | Output Shape | Param # |
|---|---|---|
| input_7 (InputLayer) | (None, 1024, 1024, 1) | 0 |
| conv2d_37 (Conv2D) | (None, 1024, 1024, 16) | 160 |
| max_pooling2d_13 (MaxPooling | (None, 512, 512, 16) | 0 |
| conv2d_38 (Conv2D) | (None, 512, 512, 8) | 1160 |
| max_pooling2d_14 (MaxPooling | (None, 256, 256, 8) | 0 |
| conv2d_39 (Conv2D) | (None, 256, 256, 8) | 584 |
| conv2d_40 (Conv2D) | (None, 256, 256, 8) | 584 |
| up_sampling2d_13 (UpSampling | (None, 512, 512, 8) | 0 |
| conv2d_41 (Conv2D) | (None, 512, 512, 16) | 1168 |
| up_sampling2d_14 (UpSampling | (None, 1024, 1024, 16) | 0 |
| conv2d_42 (Conv2D) | (None, 1024, 1024, 1) | 145 |

Total params: 3,801
Trainable params: 3,801
Non-trainable params: 0

Figura 4.9: Autoencoder separando RGB de 11 capas con imagenes de 1024x1024

3. El siguiente paso fue dividir las imágenes en trozos consecutivos, de esta forma reduciríamos el tiempo de procesado y habría un mayor numero de ejemplos para que nuestra red aprenda. En la figura 4.10 podemos ver un ejemplo de como fue la división en esta prueba.

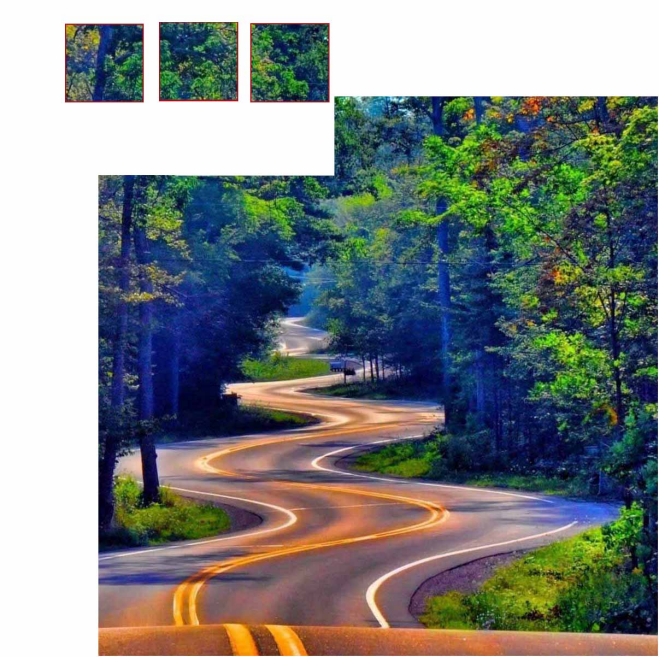


Figura 4.10: Ejemplo de como se dividió la imagen

La arquitectura es similar a las anteriores variando el tamaño de entrada y salida que dependera del tamaño que escojamos para dividir los trozos, en este caso tenemos un ejemplo con trozos de 64 píxeles 4.11.

| Layer (type) | Output Shape | Param # |
|---|---|---|
| input_2 (InputLayer) | (None, 64, 64, 3) | 0 |
| conv2d_7 (Conv2D) | (None, 64, 64, 16) | 448 |
| max_pooling2d_3 (MaxPooling2 | (None, 32, 32, 16) | 0 |
| conv2d_8 (Conv2D) | (None, 32, 32, 8) | 1160 |
| max_pooling2d_4 (MaxPooling2 | (None, 16, 16, 8) | 0 |
| conv2d_10 (Conv2D) | (None, 16, 16, 8) | 584 |
| up_sampling2d_3 (UpSampling2 | (None, 32, 32, 8) | 0 |
| conv2d_11 (Conv2D) | (None, 32, 32, 16) | 1168 |
| up_sampling2d_4 (UpSampling2 | (None, 64, 64, 16) | 0 |
| conv2d_12 (Conv2D) | (None, 64, 64, 3) | 435 |

Total params: 3,795
Trainable params: 3,795
Non-trainable params: 0

Figura 4.11: Autoencoder de 11 capas con imagenes de 64x64

A la hora de reconstruir la imagen a partir de los trozos procesados, los bordes de cada trozo quedaban sin procesar, por lo que al reconstruir la imagen quedaba ”a cuadros”. Podemos ver un ejemplo a continuación (figura 4.12).

Figura 4.12: Imagen ”a cuadros”

4. Una solución a esto fue dividir las imágenes en trozos no consecutivos. La idea era coger los trozos de la imagen de 16x16 y quedarse con los 4 píxeles del medio, que serian los que mejor describen ese trozo. De esta forma, se dividieron las imágenes en trozos solapantes en los que nos íbamos desplazando de 2 en 2 píxeles. Así, conseguiríamos reconstruir prácticamente la imagen al completo, sin contar con los bordes, con los píxeles centrales de cada trozo.

   Para ello se comenzó por dividir las imágenes en trozos iguales (de 8,16,32 y 64 píxeles) consecutivos, consiguiendo un conjunto de entrenamiento de 360448, 90112, 22528 y 5632 imágenes respectivamente.

   La arquitectura empleada en este apartado es igual que en el anterior 4.11.

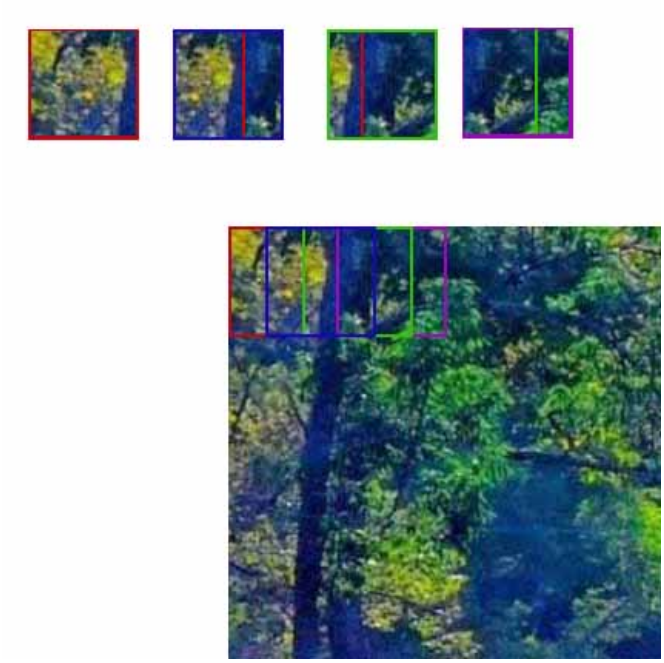


Figura 4.13: Ejemplo de como se dividió la imagen

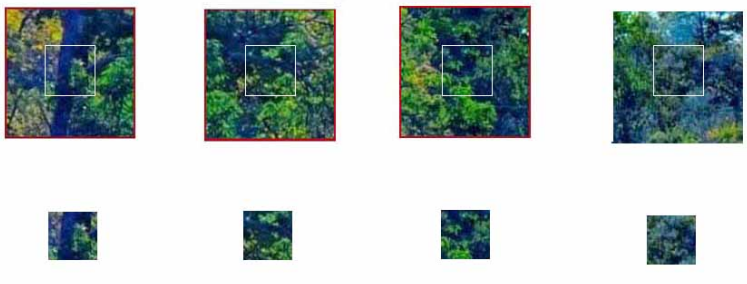


Figura 4.14: Ejemplo de como se cogieron los centros de cada trozo

5. También se realizaron pruebas de tamaño 16 y 32 en los que, al igual que en el paso 2, se entrenaba la red por capas r, g, b para ver si en este caso mejorarían los resultados. La arquitectura es similar a la del paso 2 pero con entradas y salidas igual al tamaño del trozo (16 o 32). Ejemplo con trozos de 16x16 píxeles:

| Layer (type) | Output Shape | Param # |
|---|---|---|
| input_13 (InputLayer) | [(None, 16, 16, 1)] | 0 |
| conv2d_72 (Conv2D) | (None, 16, 16, 16) | 160 |
| max_pooling2d_24 (MaxPooling | (None, 8, 8, 16) | 0 |
| conv2d_73 (Conv2D) | (None, 8, 8, 8) | 1160 |
| max_pooling2d_25 (MaxPooling | (None, 4, 4, 8) | 0 |
| conv2d_74 (Conv2D) | (None, 4, 4, 8) | 584 |
| conv2d_75 (Conv2D) | (None, 4, 4, 8) | 584 |
| up_sampling2d_24 (UpSampling | (None, 8, 8, 8) | 0 |
| conv2d_76 (Conv2D) | (None, 8, 8, 16) | 1168 |
| up_sampling2d_25 (UpSampling | (None, 16, 16, 16) | 0 |
| conv2d_77 (Conv2D) | (None, 16, 16, 1) | 145 |

Total params: 3,801
Trainable params: 3,801
Non-trainable params: 0

Figura 4.15: Autoencoder RGB con entradas y salidas de 16x16x1

6. Utilizando la arquitectura del punto 4, se realizaron pruebas para ver que resultados son mejores tras aplicar los pasos de iluminación y contraste. Las pruebas comparaban resultados aplicando primero contraste y después iluminación, iluminación y después contraste, o con un conjunto de entrada-salida en el que a la salida ya se le aplicaba la mejora de iluminación y contraste. Para realizar la comparación se usó la raíz cuadrada del error cuadrático medio de los píxeles de cada imagen como viene explicado en el apartado 4.5.

7. En la siguiente iteración, se añadió al autoencoder de la prueba 4 (en la que se entrenaba a trozos) tres clasificadores de 0 a 255 que devolviera la intensidad del píxel final, cada uno correspondiente a un canal. El primer paso fue dividir las imágenes en trozos de forma que el tamaño resultante fuera 16x16x3 y entrenar el autoencoder de esta forma (tal y como se hizo en las prueba 4). Después, una vez esté entrenado, eliminaríamos la parte del decodificador y añadiríamos en su lugar nuestro clasificador y entrenaríamos nuevamente nuestra red. Para que no se vuelva a entrenar la parte del codificador, marcaremos sus capas con ”trainable=False”, de forma que no varien los pesos y solo se entrene el clasificador. La arquitectura del autoencoder es la misma que en el apartado 4 **??**, pero la arquitectura final del codificador con el clasificador es la de la figura4.16.

| Layer (type) | Output Shape | Param # | Connected to |
|---|---|---|---|
| input_3 (InputLayer) | [(None, 16, 16, 3)] | 0 | |
| conv2d_24 (Conv2D) | (None, 16, 16, 16) | 448 | input_3[0][0] |
| batch_normalization_29 (BatchNo | (None, 16, 16, 16) | 64 | conv2d_24[0][0] |
| max_pooling2d_10 (MaxPooling2D) | (None, 8, 8, 16) | 0 | batch_normalization_29[0][0] |
| batch_normalization_30 (BatchNo | (None, 8, 8, 16) | 64 | max_pooling2d_10[0][0] |
| conv2d_25 (Conv2D) | (None, 8, 8, 8) | 1160 | batch_normalization_30[0][0] |
| batch_normalization_31 (BatchNo | (None, 8, 8, 8) | 32 | conv2d_25[0][0] |
| max_pooling2d_11 (MaxPooling2D) | (None, 4, 4, 8) | 0 | batch_normalization_31[0][0] |
| batch_normalization_32 (BatchNo | (None, 4, 4, 8) | 32 | max_pooling2d_11[0][0] |
| conv2d_26 (Conv2D) | (None, 4, 4, 8) | 584 | batch_normalization_32[0][0] |
| lambda_6 (Lambda) | (None, 4, 4) | 0 | conv2d_26[0][0] |
| lambda_7 (Lambda) | (None, 4, 4) | 0 | conv2d_26[0][0] |
| lambda_8 (Lambda) | (None, 4, 4) | 0 | conv2d_26[0][0] |
| flatten_6 (Flatten) | (None, 16) | 0 | lambda_6[0][0] |
| flatten_7 (Flatten) | (None, 16) | 0 | lambda_7[0][0] |
| flatten_8 (Flatten) | (None, 16) | 0 | lambda_8[0][0] |
| dense_12 (Dense) | (None, 128) | 2176 | flatten_6[0][0] |
| dense_13 (Dense) | (None, 128) | 2176 | flatten_7[0][0] |
| dense_14 (Dense) | (None, 128) | 2176 | flatten_8[0][0] |
| dense_15 (Dense) | (None, 256) | 33024 | dense_12[0][0] |
| dense_16 (Dense) | (None, 256) | 33024 | dense_13[0][0] |
| dense_17 (Dense) | (None, 256) | 33024 | dense_14[0][0] |

Total params: 107,984
Trainable params: 106,184
Non-trainable params: 1,800

Figura 4.16: Encoder con 3 clasificadores

Dado que los trozos que entran son de 16x16 y la salida es un único píxel, se escogió el superior izquierdo de los 4 píxeles del centro (la posición 7,7 del trozo de 16x16). La salida que nos proporciona el clasificador, es un array de 256 posiciones con la probabilidad de que la intensidad de la imagen de salida sea la de alguna de esas posiciones, por lo que si cogemos el mayor valor de este array, tendremos la probabilidad mas alta de que la intensidad final sea la de ese píxel.

Este proceso se realizo con trozos de 16x16 por temas de memoria, dado que en este caso, al ser la salida de un solo píxel, la ventana de recorte se desplazara de uno en uno sobre la imagen de 1024x1024, generando muchos mas trozos y ocupando por consiguiente más memoria. Además, se escogió este tamaño, porque en la capa de salida de nuestro codificador tendríamos un tamaño de 4x4x3, lo cual nos permite dividirlo en 3 grupos de 4 y a partir de cada grupo conseguir 3 salidas que clasifiquen en las 256 clases por

separado, una para cada capa r, g, b.

A partir de este paso, se introdujeron capas de Batch normalization. Este es un método que normaliza cada lote de datos. Normalizando los datos las distancias de los datos van de 0 a 1 y esto ayuda a la red neuronal a trabajar mejor y a tener menos problemas, pero cuando normalizamos los datos solo la capa de entrada se beneficia de esto, por lo que tendremos que poner una de estas capas después de cada capa de convolución o max-polling para tener siempre los datos normalizados. Este método también resuelve un problema llamado covariate shift. Si queremos entrenar una red neuronal necesitamos una buena cantidad de datos. Si dividimos los datos de entrenamiento en lotes más pequeños y las imágenes tienen distribuciones diferentes, por ejemplo una cierta cantidad de imágenes son de color rojo y otra cantidad son de color azul es necesario que cada lote contenga imágenes de ambas distribuciones, así el entrenamiento será más eficaz.

8. El siguiente paso fue realizar el entrenamiento de la red anterior pero separando los 3 canales RGB. Para ello se usó el autoencoder del paso 5 con un solo clasificador.

   Como en el caso anterior se escoge el mayor valor del clasificador de 0 a 255 y esa será la intensidad de alguna de las capas r, g o b del píxel, correspondiéndose con la capa r, g o b de la entrada.

| Layer (type) | Output Shape | Param # |
|---|---|---|
| input_3 (InputLayer) | [(None, 16, 16, 1)] | 0 |
| conv2d_24 (Conv2D) | (None, 16, 16, 16) | 160 |
| batch_normalization_29 (Batc | (None, 16, 16, 16) | 64 |
| max_pooling2d_10 MaxPooling2 | (None, 8, 8, 16) | 0 |
| batch_normalization_30 (Batc | (None, 8, 8, 16) | 64 |
| conv2d_25 (Conv2D) | (None, 8, 8, 8) | 1160 |
| batch_normalization_17 (Batc | (None, 8, 8, 8) | 32 |
| max_pooling2d_31(MaxPooling2 | (None, 4, 4, 8) | 0 |
| batch_normalization_32 (Batc | (None, 4, 4, 8) | 32 |
| conv2d_26 (Conv2D) | (None, 4, 4, 8) | 584 |
| flatten_1 (Flatten) | (None, 256) | 0 |
| dense_1 (Dense) | (None, 128) | 65664 |
| dense_2 (Dense) | (None, 256) | 33024 |

Total params: 100,784
Trainable params: 99,272
Non-trainable params: 1,512

Figura 4.17: Encoder con clasificador

9. En este paso es donde entran en juego las U-net explicadas en el capitulo 2.1.1.3. La arquitectura se puede ver en la figura4.18 con las conexiones entre las capas enfrentadas como se explicó.

| Layer (type) | Output Shape | Param # | Connected to |
|---|---|---|---|
| input_5 (InputLayer) | [(None, 1024, 1024, | 0 | |
| conv2d_27 (Conv2D) | (None, 512, 512, 64) | 3136 | input_5[0][0] |
| leaky_re_lu (LeakyReLU) | (None, 512, 512, 64) | 0 | conv2d_27[0][0] |
| conv2d_28 (Conv2D) | (None, 256, 256, 128 | 131200 | leaky_re_lu[0][0] |
| batch_normalization_33 (BatchNo | (None, 256, 256, 128 | 512 | conv2d_28[0][0] |
| leaky_re_lu_1 (LeakyReLU) | (None, 256, 256, 128 | 0 | batch_normalization_33[0][0] |
| conv2d_29 (Conv2D) | (None, 128, 128, 256 | 524544 | leaky_re_lu_1[0][0] |
| batch_normalization_34 (BatchNo | (None, 128, 128, 256 | 1024 | conv2d_29[0][0] |
| leaky_re_lu_2 (LeakyReLU) | (None, 128, 128, 256 | 0 | batch_normalization_34[0][0] |
| conv2d_30 (Conv2D) | (None, 64, 64, 512) | 2097664 | leaky_re_lu_2[0][0] |
| batch_normalization_35 (BatchNo | (None, 64, 64, 512) | 2048 | conv2d_30[0][0] |
| leaky_re_lu_3 (LeakyReLU) | (None, 64, 64, 512) | 0 | batch_normalization_35[0][0] |
| conv2d_31 (Conv2D) | (None, 32, 32, 512) | 4194816 | leaky_re_lu_3[0][0] |
| batch_normalization_36 (BatchNo | (None, 32, 32, 512) | 2048 | conv2d_31[0][0] |
| leaky_re_lu_4 (LeakyReLU) | (None, 32, 32, 512) | 0 | batch_normalization_36[0][0] |
| conv2d_32 (Conv2D) | (None, 16, 16, 512) | 4194816 | leaky_re_lu_4[0][0] |
| activation (Activation) | (None, 16, 16, 512) | 0 | conv2d_32[0][0] |
| conv2d_transpose (Conv2DTranspo | (None, 32, 32, 512) | 4194816 | activation[0][0] |
| batch_normalization_37 (BatchNo | (None, 32, 32, 512) | 2048 | conv2d_transpose[0][0] |
| dropout (Dropout) | (None, 32, 32, 512) | 0 | batch_normalization_37[0][0] |
| concatenate (Concatenate) | (None, 32, 32, 1024) | 0 | dropout[0][0]<br>leaky_re_lu_4[0][0] |
| activation_1 (Activation) | (None, 32, 32, 1024) | 0 | concatenate[0][0] |
| conv2d_transpose_1 (Conv2DTrans | (None, 64, 64, 512) | 8389120 | activation_1[0][0] |
| batch_normalization_38 (BatchNo | (None, 64, 64, 512) | 2048 | conv2d_transpose_1[0][0] |
| concatenate_1 (Concatenate) | (None, 64, 64, 1024) | 0 | batch_normalization_38[0][0]<br>leaky_re_lu_3[0][0] |
| activation_2 (Activation) | (None, 64, 64, 1024) | 0 | concatenate_1[0][0] |
| conv2d_transpose_2 (Conv2DTrans | (None, 128, 128, 256 | 4194560 | activation_2[0][0] |
| batch_normalization_39 (BatchNo | (None, 128, 128, 256 | 1024 | conv2d_transpose_2[0][0] |
| concatenate_2 (Concatenate) | (None, 128, 128, 512 | 0 | batch_normalization_39[0][0]<br>leaky_re_lu_2[0][0] |
| activation_3 (Activation) | (None, 128, 128, 512 | 0 | concatenate_2[0][0] |
| conv2d_transpose_3 (Conv2DTrans | (None, 256, 256, 128 | 1048704 | activation_3[0][0] |
| batch_normalization_40 (BatchNo | (None, 256, 256, 128 | 512 | conv2d_transpose_3[0][0] |
| concatenate_3 (Concatenate) | (None, 256, 256, 256 | 0 | batch_normalization_40[0][0]<br>leaky_re_lu_1[0][0] |
| activation_4 (Activation) | (None, 256, 256, 256 | 0 | concatenate_3[0][0] |
| conv2d_transpose_4 (Conv2DTrans | (None, 512, 512, 64) | 262208 | activation_4[0][0] |
| batch_normalization_41 (BatchNo | (None, 512, 512, 64) | 256 | conv2d_transpose_4[0][0] |
| concatenate_4 (Concatenate) | (None, 512, 512, 128 | 0 | batch_normalization_41[0][0]<br>leaky_re_lu[0][0] |
| activation_5 (Activation) | (None, 512, 512, 128 | 0 | concatenate_4[0][0] |
| conv2d_transpose_5 (Conv2DTrans | (None, 1024, 1024, 3 | 6147 | activation_5[0][0] |
| activation_6 (Activation) | (None, 1024, 1024, 3 | 0 | conv2d_transpose_5[0][0] |

Total params: 29,253,251
Trainable params: 29,247,491
Non-trainable params: 5,760

Figura 4.18: U-net

## 4.5 Resultados

Para comparar la calidad de los resultados, se usó la raíz cuadrada del error cuadrático medio(ECM) entre la imagen de salida y la imagen de salida ideal (la cual disponemos de ella previamente). Este error se calcula elevando al cuadrado la diferencia de los píxeles en la misma posición de ambas imágenes, y calculando la media de todos esos valores, finalmente se hace la raíz cuadrada de ese valor. Cuanto menor sea este valor, más se ajustaría la imagen de salida a la salida ideal.

Las imágenes de la figura 4.19 se corresponde a una de las imágenes ideales con las que se va a hacer la comparación.

Figura 4.19: Imagen deseada

Tanto las imágenes resultantes como las dos esperadas irán adjuntadas en este proyecto como un archivo comprimido(".zip") para que se pueda apreciar mejor la calidad de los resultados y su mejora a lo largo de las pruebas.

Los resultados de la primera prueba se pueden ver en la figura 4.20. La calidad de estas imágenes es muy baja y se ve borrosa.

Figura 4.20: Imagen de salida de la prueba 1

Los resultados de la segunda prueba se pueden ver en la figura 4.21. La calidad sigue siendo bastante mala.

Figura 4.21: Imagen de salida de la prueba 2

Los resultados de la prueba 3, donde cogemos trozos consecutivos, se ven ”a cuadros”, como en la figura 4.22.

Figura 4.22: Imagen de salida de la prueba 3

Las siguientes 4 imágenes se corresponden a las salidas de la prueba 4, en la que la imagen se dividió en trozos solapantes. Se empezó por trozos de 64x64 y se fue reduciendo hasta llegar a 8x8.

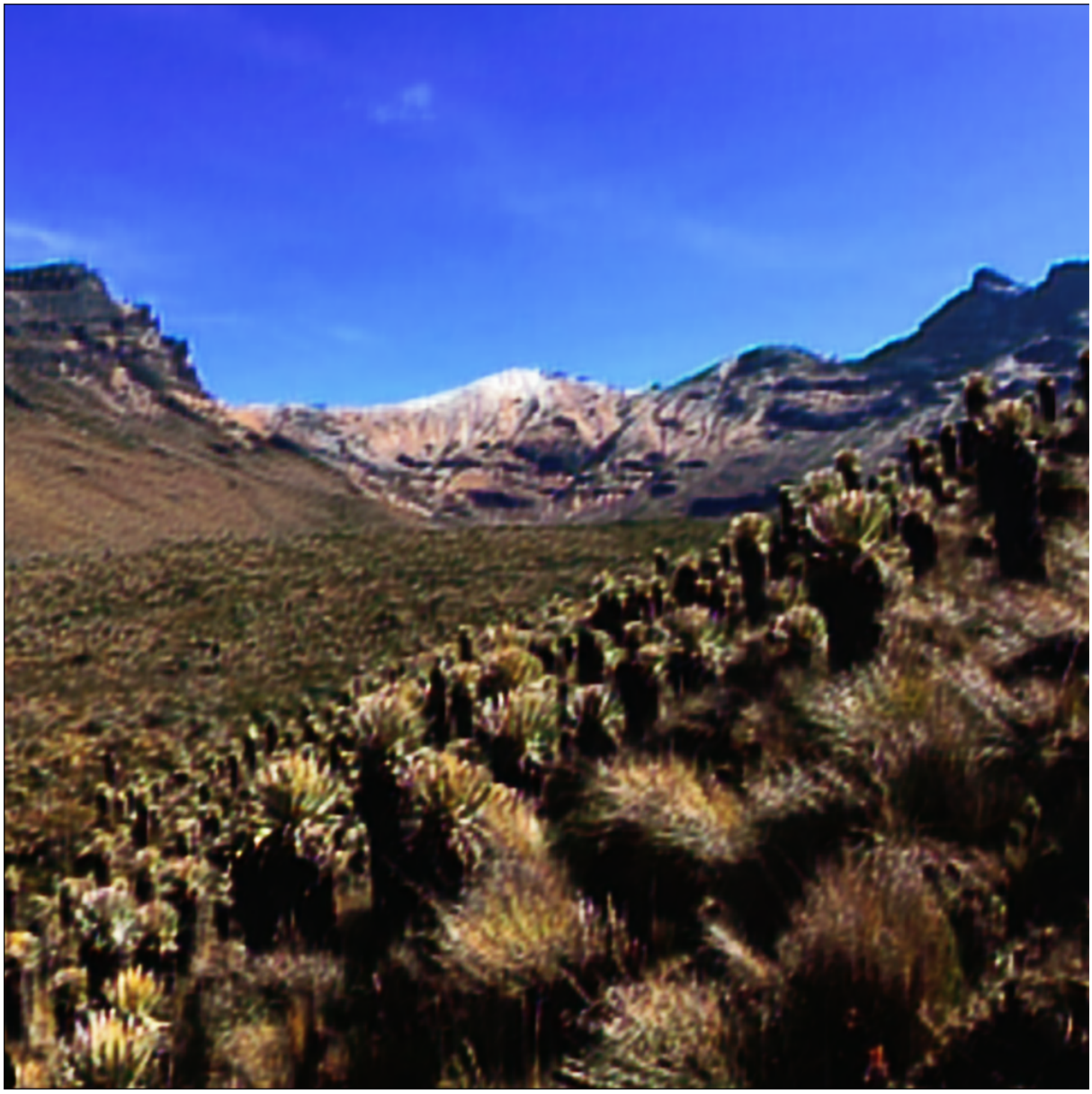


Figura 4.23: Imagen de salida de la prueba 4 con trozos de 64x64

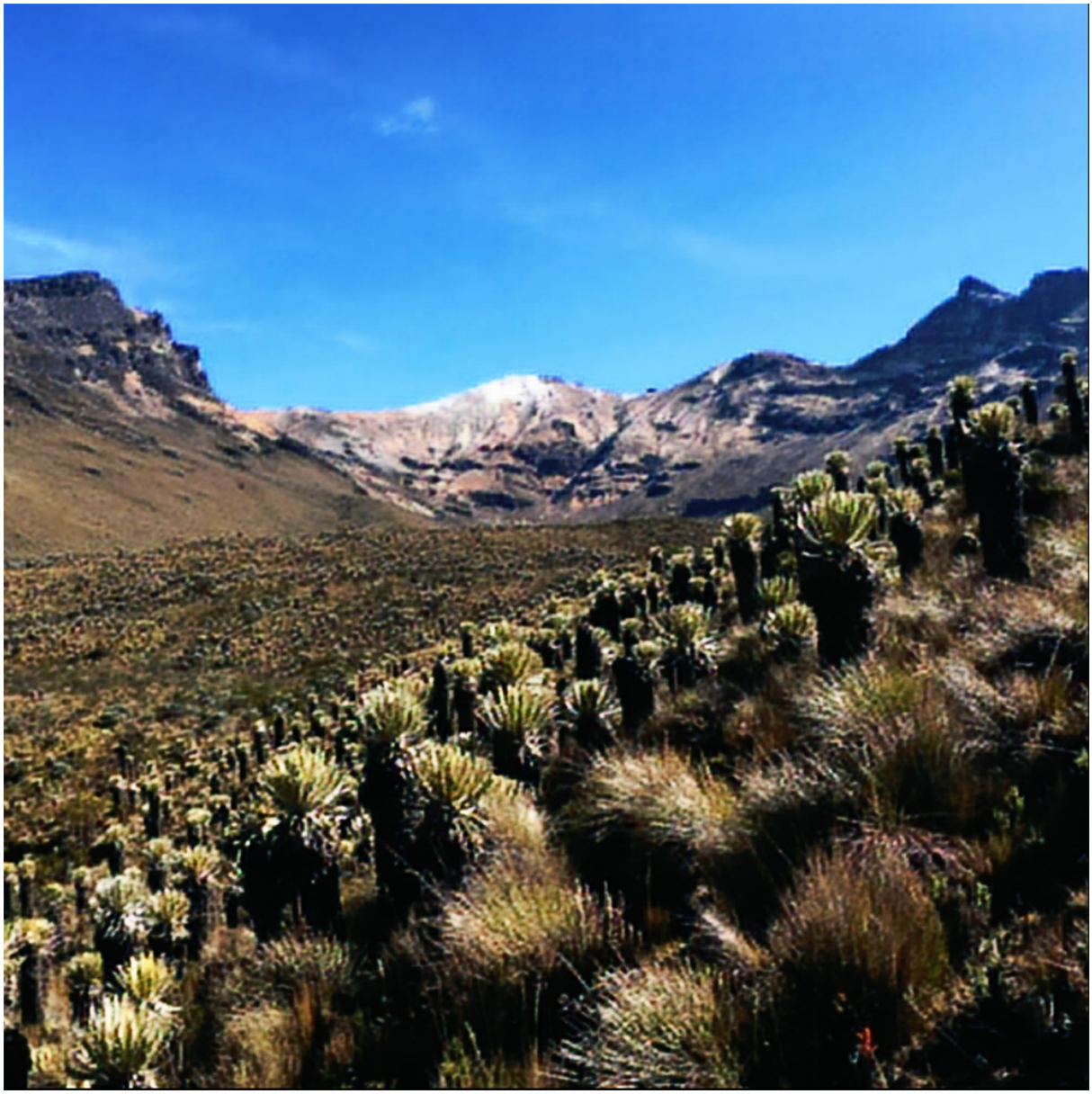


Figura 4.24: Imagen de salida de la prueba 4 con trozos de 32x32

Figura 4.25: Imagen de salida de la prueba 4 con trozos de 16x16

Figura 4.26: Imagen de salida de la prueba 4 con trozos de 8x8

Las siguientes imágenes se corresponden a la prueba 5, en la que se entrena el mismo autoencoder que en la 4 pero esta vez por capas r, g, b. Los resultados son para la división trozos de 32 y 16 píxeles de ancho y alto

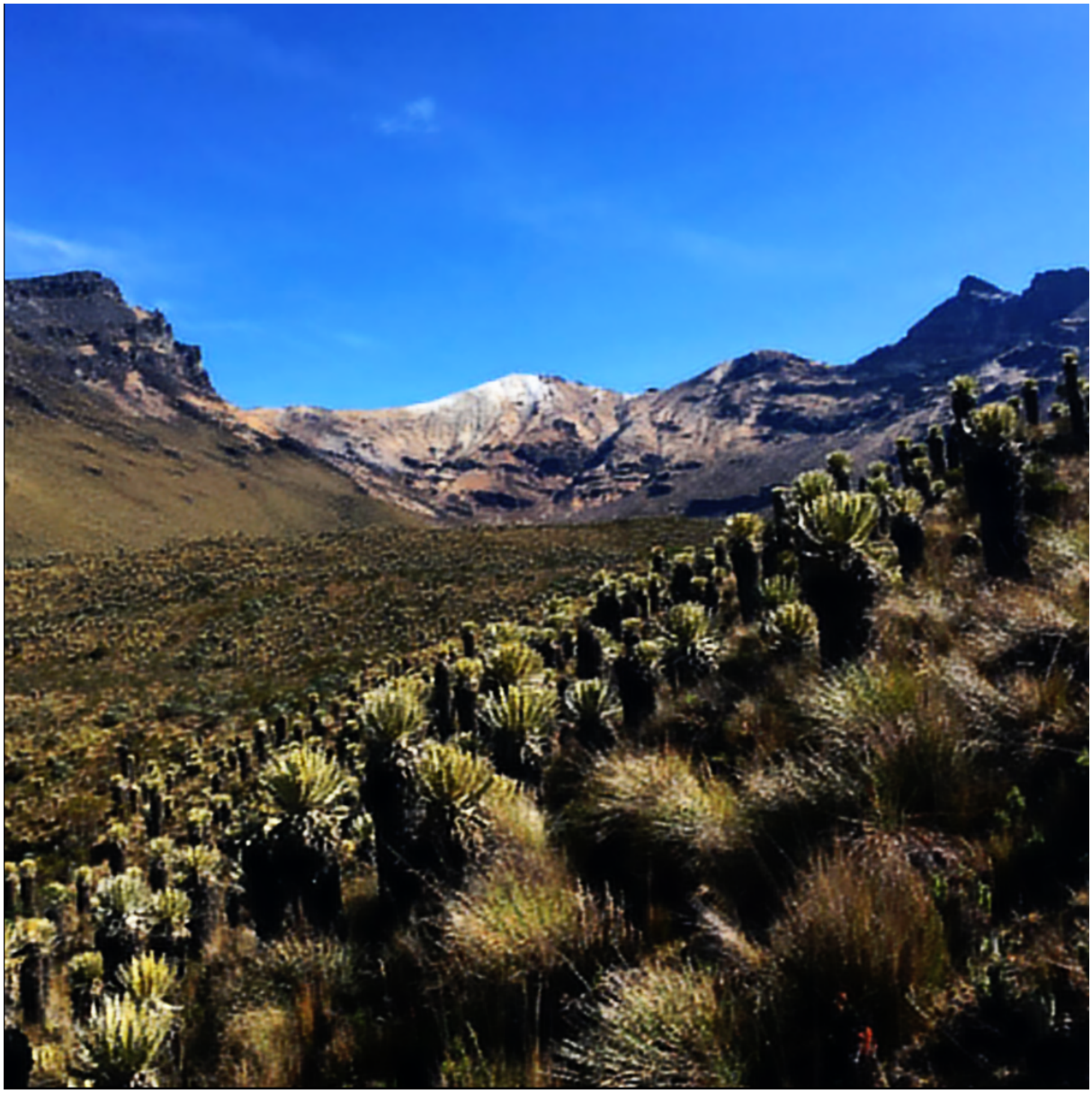


Figura 4.27: Imagen de salida de la prueba 5 con trozos de 32x32

Figura 4.28: Imagen de salida de la prueba 5 con trozos de 16x16

A continuación se muestran las tablas correspondientes al punto 6 en la que se comparan los resultados para ver cual se adecua más a nuestro resultado deseado. Las 4 primeras tablas se corresponden a la prueba 4 y la ultima tabla a la prueba 5. En este último caso solo se entrenó aplicando iluminación y contraste a la vez, pues fueron los mejores resultados de la prueba 4.

| 64x64 | Ilu->Cont | Cont->Ilu | Ilu+Cont |
|---|---|---|---|
| Img1 | 108.214 | 108.040 | 106.919 |
| Img2 | 84.602 | 84.151 | 80.702 |
| Img3 | 24.018 | 21.533 | 17.796 |
| Img4 | 32.432 | 30.742 | 26.722 |
| Img5 | 58.271 | 58.076 | 56.326 |
| Img6 | 49.873 | 49.956 | 43.777 |
| Img7 | 54.486 | 54.127 | 51.926 |
| Img8 | 60.348 | 60.822 | 55.843 |
| media | 59.031 | 58.431 | 55.002 |

Cont = Contraste
Ilu = Iluminacion

Figura 4.29: Tabla de ECM prueba 4 con trozos 64x64

| 32x32 | Ilu->Cont | Cont->Ilu | Ilu+Cont |
|---|---|---|---|
| Img1 | 76.700 | 80.727 | 76.588 |
| Img2 | 84.250 | 80.749 | 61.483 |
| Img3 | 30.740 | 28.251 | 14.485 |
| Img4 | 48.190 | 41.499 | 21.525 |
| Img5 | 49.295 | 51.359 | 41.939 |
| Img6 | 58.776 | 56.921 | 34.451 |
| Img7 | 60.050 | 58.387 | 40.186 |
| Img8 | 65.069 | 64.317 | 43.244 |
| media | 59.134 | 58.026 | 41.738 |

Figura 4.30: Tabla de ECM prueba 4 con trozos 32x32

| 16x16 | Ilu->Cont | Cont->Ilu | Ilu+Cont |
|---|---|---|---|
| Img1 | 62.268 | 61.964 | 53.904 |
| Img2 | 68.713 | 76.909 | 56.165 |
| Img3 | 25.243 | 33.225 | 14.657 |
| Img4 | 32.540 | 45.227 | 17.738 |
| Img5 | 41.049 | 45.403 | 34.689 |
| Img6 | 51.027 | 57.570 | 36.769 |
| Img7 | 46.301 | 55.657 | 36.765 |
| Img8 | 54.377 | 57.201 | 37.889 |
| | | | |
| media | 47.690 | 54.145 | 36.072 |

Figura 4.31: Tabla de ECM prueba 4 con trozos 16x16

| 8x8 | Ilu->Cont | Cont->Ilu | Ilu+Cont |
|---|---|---|---|
| Img1 | 44.261 | 106.919 | 38.550 |
| Img2 | 48.107 | 80.702 | 42.907 |
| Img3 | 16.432 | 17.796 | 17.639 |
| Img4 | 22.603 | 26.722 | 21.040 |
| Img5 | 28.453 | 56.326 | 26.930 |
| Img6 | 30.112 | 43.777 | 30.263 |
| Img7 | 31.605 | 51.926 | 30.123 |
| Img8 | 30.108 | 55.843 | 31.102 |
| | | | |
| media | 31.460 | 55.002 | 29.819 |

Figura 4.32: Tabla de ECM prueba 4 con trozos 8x8

Prueba 5 ILU+CONT

| | 32x32 | 16x16 |
|---|---|---|
| Img1 | 75.548 | 53.564 |
| Img2 | 59.621 | 52.262 |
| Img3 | 13.278 | 13.327 |
| Img4 | 19.866 | 16.801 |
| Img5 | 40.925 | 33.685 |
| Img6 | 31.966 | 35.184 |
| Img7 | 38.699 | 36.063 |
| Img8 | 39.273 | 37.887 |
| | | |
| media | 39.897 | 34.847 |

Figura 4.33: Tabla de ECM prueba 5 con trozos 16x16 y 32x32

Imágenes resultado de la prueba 7 en la que usamos un autoencoder con 3 clasificadores para trabajar con los 3 canales a la vez:

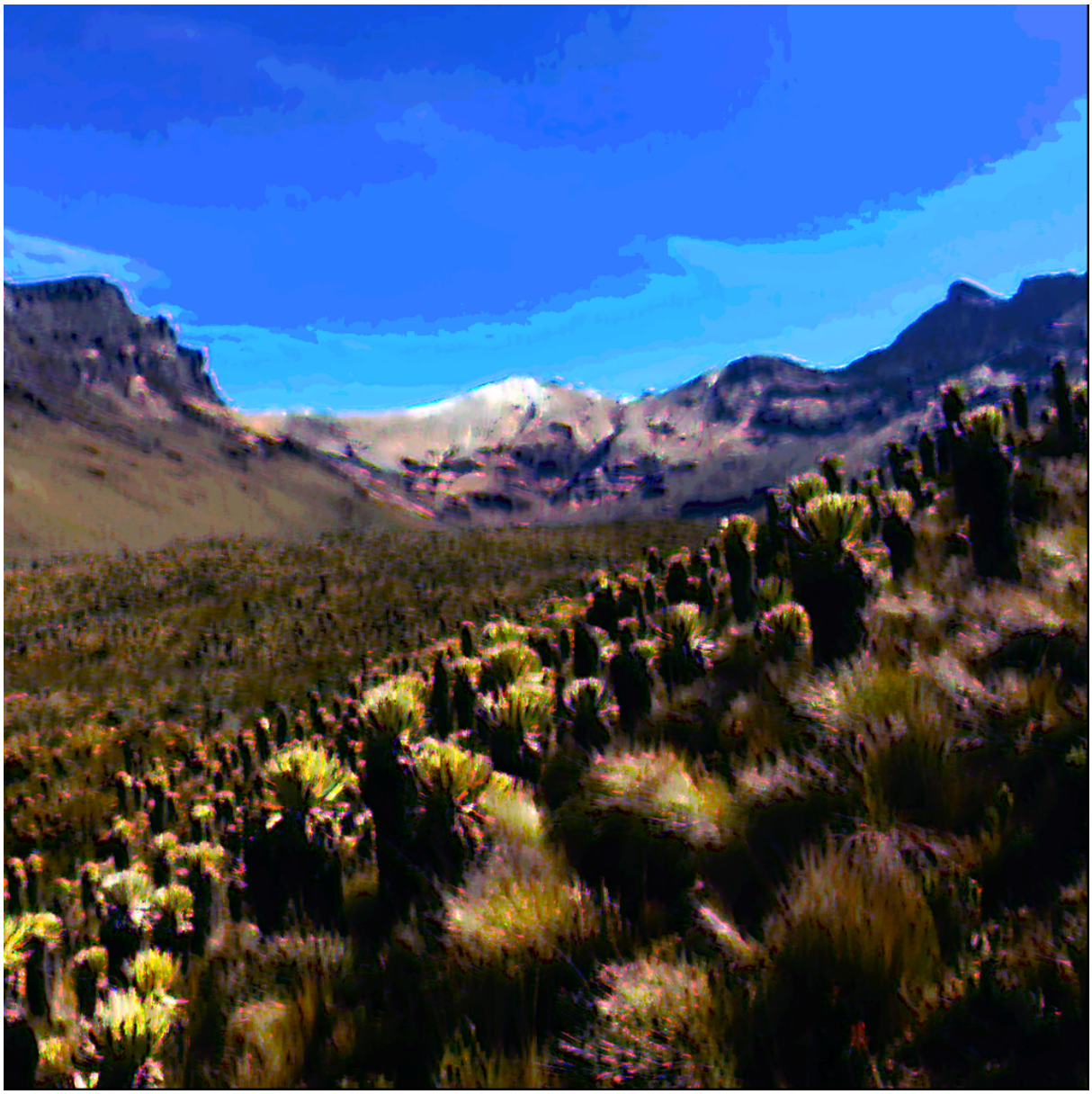


Figura 4.34: Imagen de salida de la prueba 7

Autoencoder con clasificador 16x16 con 3 salidas

| 16x16 | Ilu->Cont | Cont->Ilu | Ilu+Cont |
|---|---|---|---|
| Img1 | 57.907 | 58.815 | 54.304 |
| Img2 | 62.735 | 72.652 | 62.512 |
| Img3 | 18.337 | 21.525 | 18.162 |
| Img4 | 21.970 | 24.046 | 20.709 |
| Img5 | 38.100 | 40.188 | 38.658 |
| Img6 | 45.178 | 46.848 | 48.299 |
| Img7 | 42.991 | 45.361 | 44.023 |
| Img8 | 50.782 | 51.381 | 51.818 |
| | | | |
| media | 42.250 | 45.102 | 42.311 |

Figura 4.35: Tabla de ECM prueba 7

Imágenes resultado de la prueba 8 cuya arquitectura es similar a la de la prueba 7, aunque con un solo clasificador y separando los canales r,g,b:

Figura 4.36: Imagen de salida de la prueba 8

Autoencoder con clasificador 16x16 por canales RGB

| 16x16 | Ilu->Cont | Cont->Ilu | Ilu+Cont |
|---|---|---|---|
| Img1 | 50.688 | 51.048 | 41.124 |
| Img2 | 50.599 | 52.758 | 45.768 |
| Img3 | 13.437 | 15.548 | 12.085 |
| Img4 | 18.173 | 20.586 | 15.174 |
| Img5 | 33.331 | 34.549 | 27.563 |
| Img6 | 38.330 | 40.164 | 31.410 |
| Img7 | 31.936 | 41.329 | 31.579 |
| Img8 | 40.717 | 38.891 | 30.775 |
| | | | |
| media | 33.401 | 32.984 | 26.150 |

Figura 4.37: Tabla de ECM prueba 8 con trozos

Resultados prueba 9 en la que probamos la red U-net:

Figura 4.38: Imagen de salida de la prueba 9

U-net

| | U-net |
|---|---|
| Img1 | 18.170 |
| Img2 | 75.527 |
| Img3 | 55.651 |
| Img4 | 52.249 |
| Img5 | 35.276 |
| Img6 | 76.201 |
| Img7 | 47.207 |
| Img8 | 64.809 |

| | |
|---|---|
| media | 47.231 |

Figura 4.39: Tabla de ECM prueba 9 con trozos

## 4.6 Consideraciones

Como podemos observar en los resultados de las dos primeras pruebas, el autoencoder funciona mejor si se entrena con la imagen entera y separando las capas r, g, b. Esto se ve reflejado en las pruebas 4 y 5, en las que, al usar el mismo autoencoder pero dividiendo la imagen en trozos, sigue dando mejores resultados el no separar las imágenes en capas.

Sin embargo, al añadir un clasificador cambia el asunto. En el caso de los trozos sin separación de capas, tenemos que separar los trozos reducidos por el codificador en 3 para producir 3 salidas, una por capa del píxel de salida, y esto al autoencoder le perjudica porque no es capaz de asociar correctamente a partir de los trozos codificados 3 salidas diferentes. Por eso, en el caso de las pruebas 7 y 8, los resultados de la prueba 8 son mucho mejores, mejorando los de los pasos 4 y 5.

Por último, la red U-net, necesita ser entrenada por la imagen al completo, puesto que si entrenamos a trozos las concatenaciones de capas no nos aportarían apenas contexto para reconstruir la imagen, y este es uno de sus inconvenientes. Ademas de no permitir un conjunto de entrenamiento amplio al no poder trocear las imágenes, el hecho de tener tantas capas, por mucho que estén conectadas, no ayudan lo suficiente como para lograr una buena calidad en la salida.

## 4.7 Integracion en el NCS2

Hay muchos de marcos de aprendizaje profundo utilizados en la industria, pero los que admite el Intel Compute Stick 2 son Caffe, TensorFlow, MXNet, Kaldi y ONXX.

A partir de este momento llamaremos modelo a nuestra RNA para que no haya confusión con los nombres que usa Intel para describir sus métodos.

A continuación se explicará el flujo de trabajo de nuestro Intel Compute Stick:

1. Generación del modelo entrenado en uno de los marcos admitidos.
2. Configurar Model Optimizer para el marco específico (con el que entrenó su modelo).
3. Ejecutar Model Optimizer para producir una Representación intermedia (IR) optimizada del modelo.
4. Probar el modelo en el formato IR utilizando el motor de inferencia.
5. Integrar Inference Engine en su aplicación para implementar el modelo en el entorno de destino.

Como se puede ver en la figura 4.40 la ejecución en el Intel Movidius se divide en dos partes importantes: la encargada de optimizar los modelos (optimizador, color verde) y la parte encargada de predecir las respuestas (inferenciador, color azul).

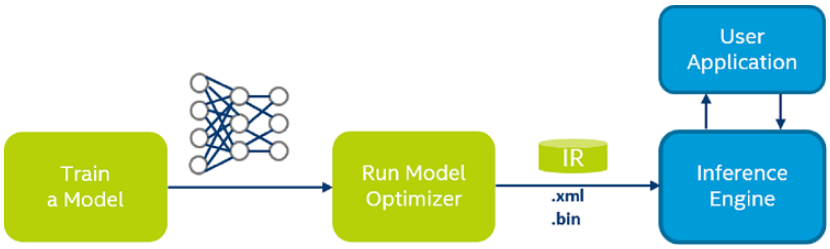



Figura 4.40: Flujo de trabajo en el Intel Compute Stick

### 4.7.1 Generación del modelo entrenado

Para generar un modelo entrenado usaremos TensorFlow y dentro de este usaremos la librería Keras, dado que al ser una API de más alto nivel nos proporciona una serie de funciones para crear, entrenar y guardar modelos de una manera sencilla y en cuestión de pocas líneas, sin necesidad de ajustar los pesos a mano. El lenguaje de programación usado será Python.

Usando Keras, crearemos un modelo y lo entrenaremos con conjuntos de imágenes (entrada y salida), usando autoencoders para extraer diferentes características que puedan servirle a nuestro modelo para trabajar con futuras imágenes.

Para guardar el modelo usaremos la función save() de nuestro modelo para que almacene todos los pesos y relaciones de los nodos y le pasaremos una ruta donde queramos almacenarlo.

#### 4.7.1.1 Congelar el modelo

Una vez tengamos el modelo creado con TensorFlow, tenemos que congelar el modelo antes de optimizarlo. Para ello leeremos el modelo y eliminaremos cualquiera operación o etiqueta que no sea necesaria para la inferencia. A este proceso se le llama congelar el modelo y es recomendable ya que permite eliminar muchas operaciones relacionadas con la carga y el almacenamiento de las variables.

Antes de nada debemos establecer la fase de aprendizaje a 0 con "tf.keras.backend.set_learning_phase(0)".

Después usamos estos dos métodos recorriendo el grafo:

->tf.graph_util.remove_training_node: Hay nodos como Identity y CheckNumerics que solo son útiles durante el entrenamiento, y pueden eliminarse en gráficos que se usarán solo para inferencia.

->tf.graph_util.convert_variables_to_constants: Si tiene un gráfico entrenado que contiene operaciones de Variable, puede ser conveniente convertirlas todas a operaciones de Constantes que tengan los mismos valores.

Por último guardamos nuestro modelo congelado con el nombre nombreDeNuestroModelo.pb usando la función:

->graph_io.write_graph(graphdef_frozen, save_pb_dir, save_pb_name,

as_text=save_pb_as_text)

En el caso de que hayamos guardado nuestro modelo con save_weights en vez de save_model, tendremos que añadir la siguiente linea justo después de cargar nuestro modelo con load_weights:

tf.keras.backend.get_session().run(tf.initialize_all_variables())

Para que inicie las variables y así poder convertirlas en constantes.

La principal diferencia con el modelo previo a la congelación es que al convertir en constantes las variables ya no se puede volver a entrenar.

### 4.7.2 Configurar Model Optimizer para el marco específico

Model Optimizer carga un modelo en la memoria, lo lee, construye la representación interna del modelo, lo optimiza y produce la Representación intermedia (RI).

El proceso asume que se tiene un modelo de red capacitado utilizando uno de los marcos compatibles. El flujo de trabajo de Model Optimizer es el siguiente:

- Configure Model Optimizer para uno de los marcos de aprendizaje profundo compatibles que se usaron para entrenar el modelo. Para ello, vaya a la ruta: INSTALL_DIR/ deployment_tools/model_optimizer/install_prerequisites donde lo instalo y ejecute: "./install_prerequisites.sh".
- Proporcione como entrada una red capacitada que contenga una cierta topología de red y los pesos y sesgos ajustados (con algunos parámetros opcionales).
- Ejecute Model Optimizer para realizar optimizaciones específicas del modelo (por ejemplo, fusión horizontal de ciertas capas de red).
- Model Optimizer produce como salida una Representación intermedia (IR) de la red que se utiliza como entrada para el motor de inferencia en todos los objetivos. El IR es un par de archivos que describen todo el modelo:
  - .xml: El archivo de topología: un archivo XML que describe la topología de la red
  - .bin: El archivo de datos: contiene los datos binarios de pesos y sesgos
- Los archivos de representación intermedia (IR) se pueden leer, cargar e inferir con Inference Engine.

La representación intermedia es el único formato que acepta el motor de inferencia.

### 4.7.3 Producir la representación intermedia optimizada

Para la creación de la IR necesitaremos inicializar nuestras variables OpenVino yendo a nuestra ruta de instalación:

cd /intel/computer_vision_sdk/bin/

y ejecutar el comando: ./setupvars.sh

El siguiente paso es utilizar nuestro archivo model optimizer (mo.py) situado en la carpeta /intel/openvino/deployment_tools/model_optimizer/

El comando a ejecutar seria:

python3 mo.path –input_model pb.file –output_dir output.dir –input_shape input.shape.str –data_type FP16

pb_file es nuestro modelo congelado

input_shape es el tamaño de entrada al modelo, si no se especifica lo adquiere automáticamente del modelo, aunque a veces esto falla.

Este paso generará dos archivos frozen_model.xml y frozen_model.bin. Son la Representación Intermedia optimizada del modelo basada en la topología de red capacitada, los valores de ponderación y sesgos. Los usaremos para inferenciar.

### 4.7.4 Integrar Inference Engine en su aplicación

El motor de inferencia es una biblioteca de C ++ con un conjunto de clases de C ++ para inferir datos de entrada (imágenes) y obtener un resultado. La biblioteca de C ++ proporciona una API para leer la Representación intermedia, establecer los formatos de entrada y salida y ejecutar el modelo en los dispositivos.

Esta biblioteca contiene las clases para:

- Lea la red (CNNNetReader)
- Manipular la información de la red (CNNNetwork)
- Crea y usa los diferentes complementos (PluginDispatcher)
- Ejecutar y pasar entradas y salidas (ExecutableNetwork e InferRequest)

#### 4.7.4.1 Pasos de integración

El proceso de integración incluye los siguientes pasos:

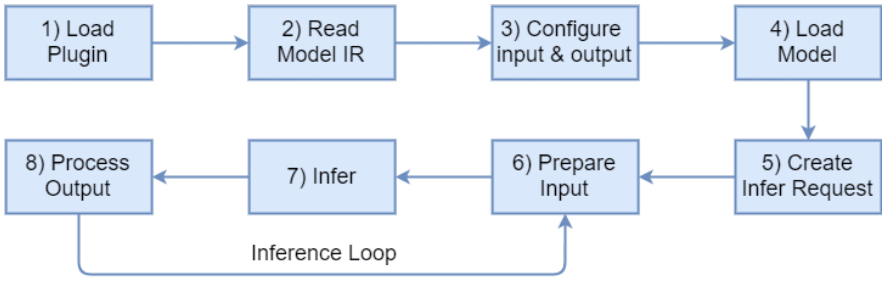



Figura 4.41: Proceso de integración del Inferenciador

1. Cargar un plugin creando una instancia de InferenceEnginePluginPtr:
   ”InferenceEnginePluginPtr engine= PluginDispatcher(pluginDirs).getSuitablePlugin(TargetDevice)” y ”InferencePlugin plugin(engine);”

2. Crear un lector de IR creando una instancia CNNNetReader y leyendo un modelo de IR creado por Model Optimizer:
   CNNNetReader network_reader;
   ”network_reader.ReadNetwork (”Model.xml”);”
   ”network_reader.ReadWeights (”Model.bin”);”

3. Configurar entrada y salida.
   red automática = network_reader.getNetwork ();

InputsDataMap input_info (network.getInputsInfo ());
OutputsDataMap output_info (network.getOutputsInfo ());

4. Cargue el modelo al plugin usando LoadNetwork():
   auto executable_network = plugin.LoadNetwork(network, );

5. Crear una solicitud de inferir :
   auto infer_request = executable_network.CreateInferRequest ();

6. Preparar la entrada.

7. Hacer inferencia llamando a los métodos StartAsync y Wait para la solicitud asíncrona:
   infer_request-> StartAsync ();
   infer_request.Wait (RESULT_READY);
   o llamando al Infer, método de solicitud síncrona:
   sync_infer_request-> Infer ();

Capítulo 5

# Conclusións

Con los resultados obtenidos podemos sacar las siguientes conclusiones:

- Los autoencoder son redes neuronales que son capaces de entrenarse rápidamente y ser flexibles para aprender cualquier tipo de curva que queramos enseñarle. Además, reconstruye las imágenes perfectamente aunque se pierda un poco de calidad.
- En cuanto a las pruebas realizadas, podemos observar por las tablas que los mejores resultados los da un autoencoder al que se le añade un clasificador y al que se le pasan como entradas las imágenes separadas en capas r, g, b. A mayores, al dividir las imágenes en trozos y capas nos proporciona un conjunto de entrenamiento mucho mayor, por lo que la red aprende de forma mas eficaz.
- El mayor problema de usar el autoencoder con el clasificador, es que te devuelve un único píxel por cada trozo de 16x16 (en nuestra prueba 8), por lo que nuestro sistema tiene que procesar muchos trozos para reconstruir la imagen píxel a píxel, pero gracias a nuestro hardware específico de Intel este problema se ve reducido.

# Bibliografía


[1] M. Feldman. (2008) Market for artificial intelligence projected to hit 36 billion by 2025. [Online]. Available: https://www.top500.org/news/market-for-artificial-intelligence-projected-to-hit-36-billion-by-2025/

[2] (2008) Tractica. [Online]. Available: https://www.tractica.com/about/overview/

[3] (2018) ¿qué es la ley de moore y para qué sirve? [Online]. Available: https://www.profesionalreview.com/2018/04/01/que-es-la-ley-de-moore-y-para-que-sirve/

[4] A. G. Serrano, *INTELIGENCIA ARTIFICIAL. Fundamentos, práctica y aplicaciones*, 2nd ed. RC LIBROS, 2016.

[5] P. N. Stuart Russell, *Artificial Intelligence: A Modern Approach*, 3rd ed. PRENTICE-HALL INTERNAC, 1994.

[6] J. R. H. V. J. Martínez, *Redes neuronales artificiales. Fundamentos, modelos y aplicaciones.* Ra-Ma, 1995.

[7] F. Berzal, *Redes Neuronales and Deep Learning.* Independently Published, 2018. [Online]. Available: https://books.google.es/books?id=TbZUvwEACAAJ

[8] F. Chollet. (2016) Building autoencoders in keras. [Online]. Available: https://blog.keras.io/building-autoencoders-in-keras.html

[9] Z. Tang, X. Peng, K. Li, and D. N. Metaxas, "Towards efficient u-nets: A coupled and quantized approach," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1–1, 2019.

[10] Google. (2019) Tensorflow. [Online]. Available: https://www.tensorflow.org

[11] F. Chollet. (2015) Keras. [Online]. Available: https://keras.io

[12] Intel. (2018) Barra de cómputo neuronal intel 2. [Online]. Available: https://software.intel.com/es-es/neural-compute-stick

[13] ——. (2018) Intel® movidius™ myriad™ x vpu. [Online]. Available: https://www.movidius.com/myriadx

[14] J. Kim, M. Kim, H. Kang, and K. Lee, "U-gat-it: Unsupervised generative attentional networks with adaptive layer-instance normalization for image-to-image translation," 2019.

[15] (2017) Probamos photolemur, un programa de procesamiento automático de imagen por inteligencia artificial. [Online]. Available: https://fotografodigital.com/noticias/probamos-photolemur-un-programa-de-procesamiento-automatico-de-imagen-por-inteligencia-artificial/

[16] Image super-resolution using deep convolutional networks. [Online]. Available: http://mmlab.ie.cuhk.edu.hk/projects/SRCNN.html

# Apéndices

## Apéndice A

# Glosario de acrónimos

**IA** *Inteligencia Artificial.*

**NCS** *Neural Compute Stick.*

**VPU** *Vision Processing Unit.*

**VLIW** *Very Long Instruction Word.*

**FPS** *Fotogramas Por Segundo.*

**API** *Interfaz de Programación de Aplicaciones o Application Programming Interface.*

**RNA** *Red Neuronal Artificial.*

**DNN** *Deep Neural Networks o redes neuronales profundas.*

**NPU** *Unidad de Procesamiento Neuronal.*

**hw** *Hardware.*

**sw** *Software.*

**PLN** *Procesamiento del Lenguaje Natural.*

**ECM** *Error cuadrático medio.*

## Apéndice B

# Glosario de términos

**Machine learning** Subcampo de las ciencias de la computación y una rama de la inteligencia artificial, cuyo objetivo es desarrollar técnicas que permitan que las computadoras aprendan.

**Deep learning** El Deep Learning lleva a cabo el proceso de Machine Learning usando una red neuronal artificial que se compone de un número de niveles jerárquicos. En el nivel inicial de la jerarquía la red aprende algo simple y luego envía esta información al siguiente nivel.

**Hardware** Conjunto de elementos físicos que constituyen un sistema informático.

**Software** Conjunto de programas que permiten a la computadora realizar determinadas tareas

**Ley de Moore** Expresa que aproximadamente cada dos años se duplica el número de transistores en un microprocesador.

**Contraste** Diferencia de intensidad de iluminación en la gama de blancos y negros o en la de colores de una imagen.

**Rendimiento** Utilidad de una cosa en relación con lo que gasta.

**Algoritmo** Conjunto ordenado de operaciones sistemáticas que permite hacer un cálculo y hallar la solución de un tipo de problemas.

**Procesamiento del lenguaje natural** Campo de las ciencias de la computación, inteligencia artificial y lingüística que estudia las interacciones entre las computadoras y el lenguaje humano.

**Sinapsis** Almacenan parámetros llamados ”pesos” que manipulan los datos en los cálculos.

**Codificación** Método que permite convertir un carácter de un lenguaje natural en un símbolo de otro sistema.

**Procesador** Componente electrónico donde se realizan los procesos lógicos.

**Backend** ”Motor de fondo” o parte que trabaja oculta al usuario.

**Compilador** Un compilador es un tipo de traductor que transforma un programa entero de un lenguaje de programación (llamado código fuente) a otro.

**código fuente** El código fuente de un programa informático (o software) es un conjunto de líneas de texto con los pasos que debe seguir la computadora para ejecutar un programa.

**Inferencia** Juicio o conclusión a partir de hechos, proposiciones o principios, sean generales o particulares.

**Punto flotante** Forma de notación científica usada en los computadores con la cual se pueden representar números reales extremadamente grandes y pequeños de una manera muy eficiente y compacta.

**Píxeles** Unidad básica de una imagen digitalizada en pantalla a base de puntos de color o en escala de grises.